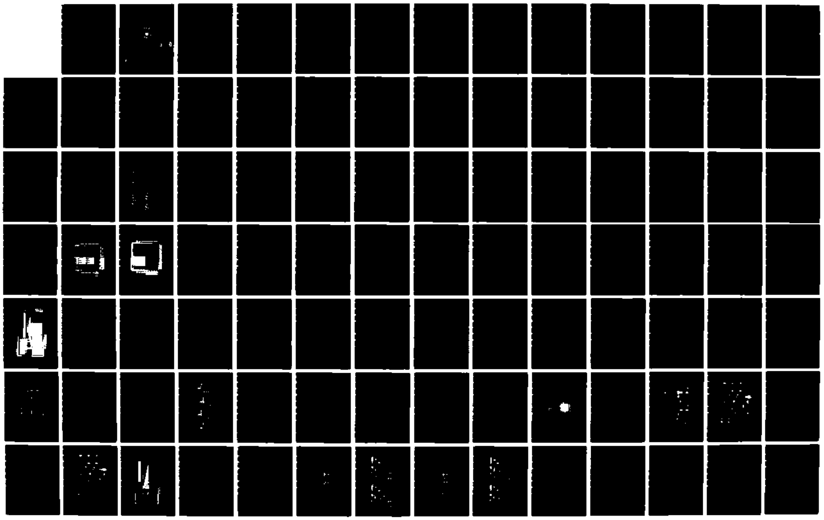
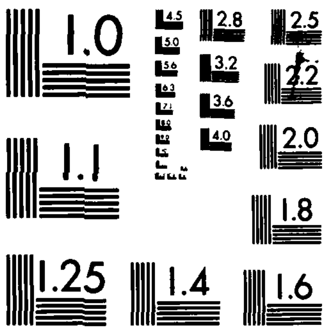
AD-A158 995 SILICON COMPILER DESIGN OF COMBINATIONAL AND PIPELINE ADDER INTEGRATED CIRCUITS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA A O FROEDE JUN 85 1/2

UNCLASSIFIED F/G 9/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

AD-A158 995

DTIC FILE COPY

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC ELECTE SEP 1 2 1985 S D A

THESIS

SILICON COMPILER DESIGN OF
COMBINATIONAL AND PIPELINE
ADDER INTEGRATED CIRCUITS

by

Alexander O. Froede, III

June 1985

Thesis Advisor: D. E. Kirk

Approved for public release; distribution is unlimited

85 9 10 024

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A158	3. RECIPIENT'S CATALOG NUMBER 995
4. TITLE (and Subtitle) Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alexander O. Froede, III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		12. REPORT DATE June 1985
		13. NUMBER OF PAGES 139
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI Design; MacPitts Silicon Compiler; Pipeline Adder; Block Carry Look Ahead Addition; Data-Path; Weinberger Array

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The architecture and structures used by the MacPitts silicon compiler to design integrated circuits are described, and the capabilities and limitations of the compiler are discussed. The performance of several combinational and pipeline adders designed by MacPitts and a hand-crafted pipeline adder are compared. Several different MacPitts design errors are documented. Tutorial material is presented to aid in using the MacPitts interpreter and to illustrate timing analysis of MacPitts-designed circuits using the program Crystal.

[illegible]

S/N 0102-LF-014-6601

Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits

by

Alexander O. Froede, III
Captain, United States Army
B.S., University of Arizona, 1978

A-1

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

Author: Alexander O. Froede III
Alexander O. Froede, III

Approved by: [illegible]
D.E. Kirk, Thesis Advisor

[illegible]
H.H. Loomis, Second Reader

[illegible]
H. B. Rigas, Department of Electrical and Computer Engineering

[illegible]
J.N. Dyer, Dean of Science and Engineering

ABSTRACT

The architecture and structures used by the MacPitts silicon compiler to design integrated circuits are described, and the capabilities and limitations of the compiler are discussed. The performance of several combinational and pipeline adders designed by MacPitts and a hand-crafted pipeline adder are compared. Several different MacPitts design errors are documented. Tutorial material is presented to aid in using the MacPitts interpreter and to illustrate timing analysis of MacPitts-designed circuits using the program Crystal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The design of very large scale integrated (VLSI) circuits is a very time consuming process. To reduce the time and cost required to design VLSI circuits various silicon compilers have been developed [Ref. 1]. One of these compilers, the MacPitts silicon compiler, was developed at MIT Lincoln Laboratory in 1981-1982 [Refs. 2 and 3].

The MacPitts silicon compiler is a large and complex computer program that frees the circuit designer from having to worry about the details of the actual design and layout of the circuit. From a short program (usually less than fifty lines) that contains a *functional* description of the desired circuit, MacPitts completely designs an implementation of the VLSI chip and outputs a file in Caltech Intermediate Form (CIF) that describes the circuit. The CIF file can be used to perform a functional simulation or a timing analysis of the circuit. After verifying the functional correctness of the circuit the CIF file can be sent to a silicon foundry so that the circuit can be fabricated.

The MacPitts compiler has been used previously at the Naval Postgraduate School by Carlson [Ref. 4] to design a pipeline multiplier circuit. Carlson's thesis contains a

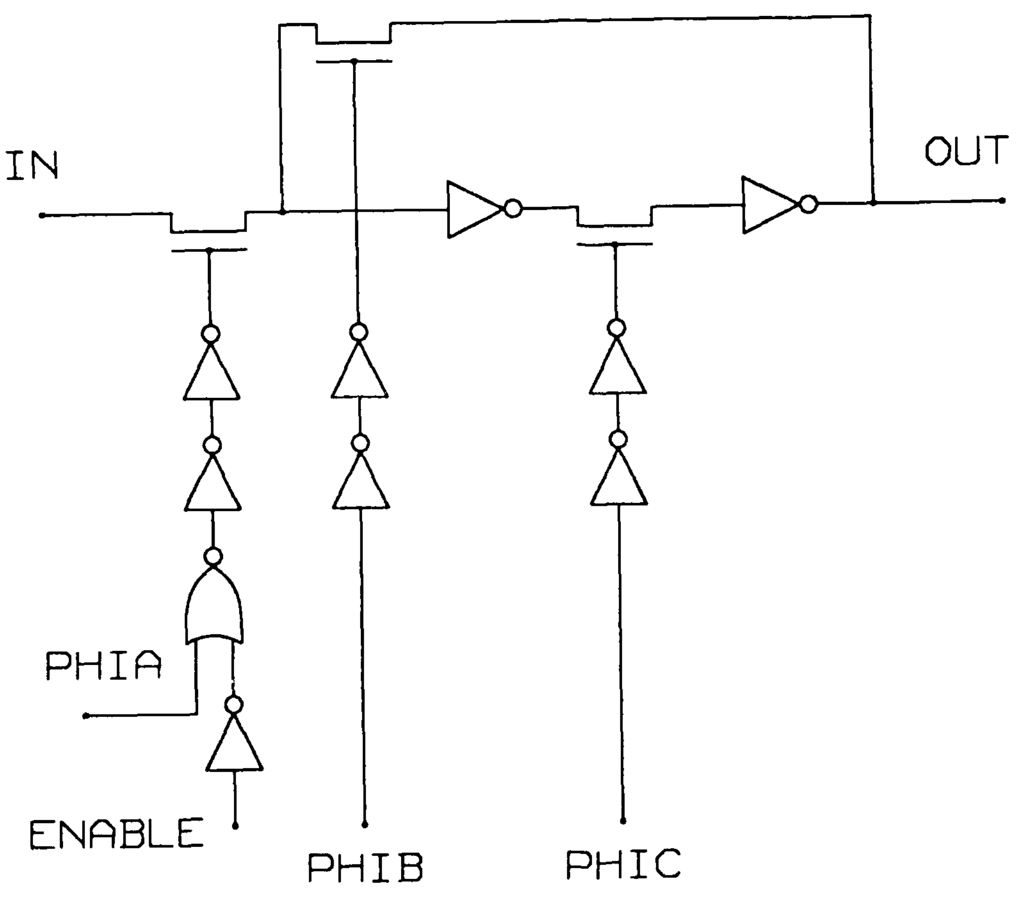


Figure 2.7. 1-Bit Register Cell

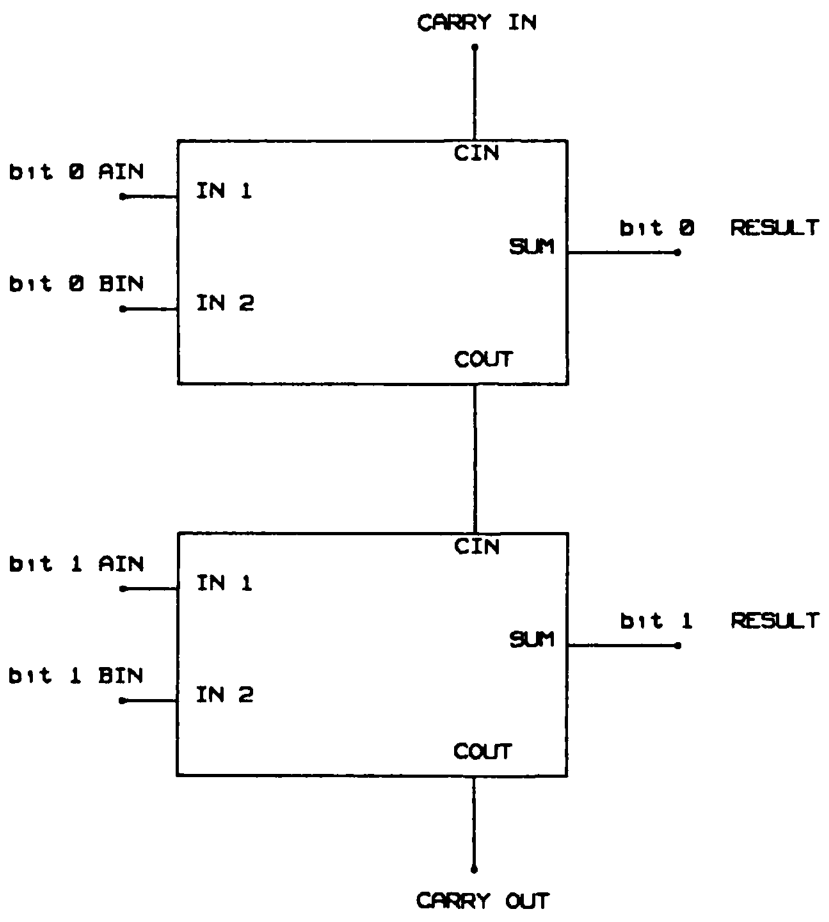


Figure 2.6. 2-Bit Full Adder Circuit

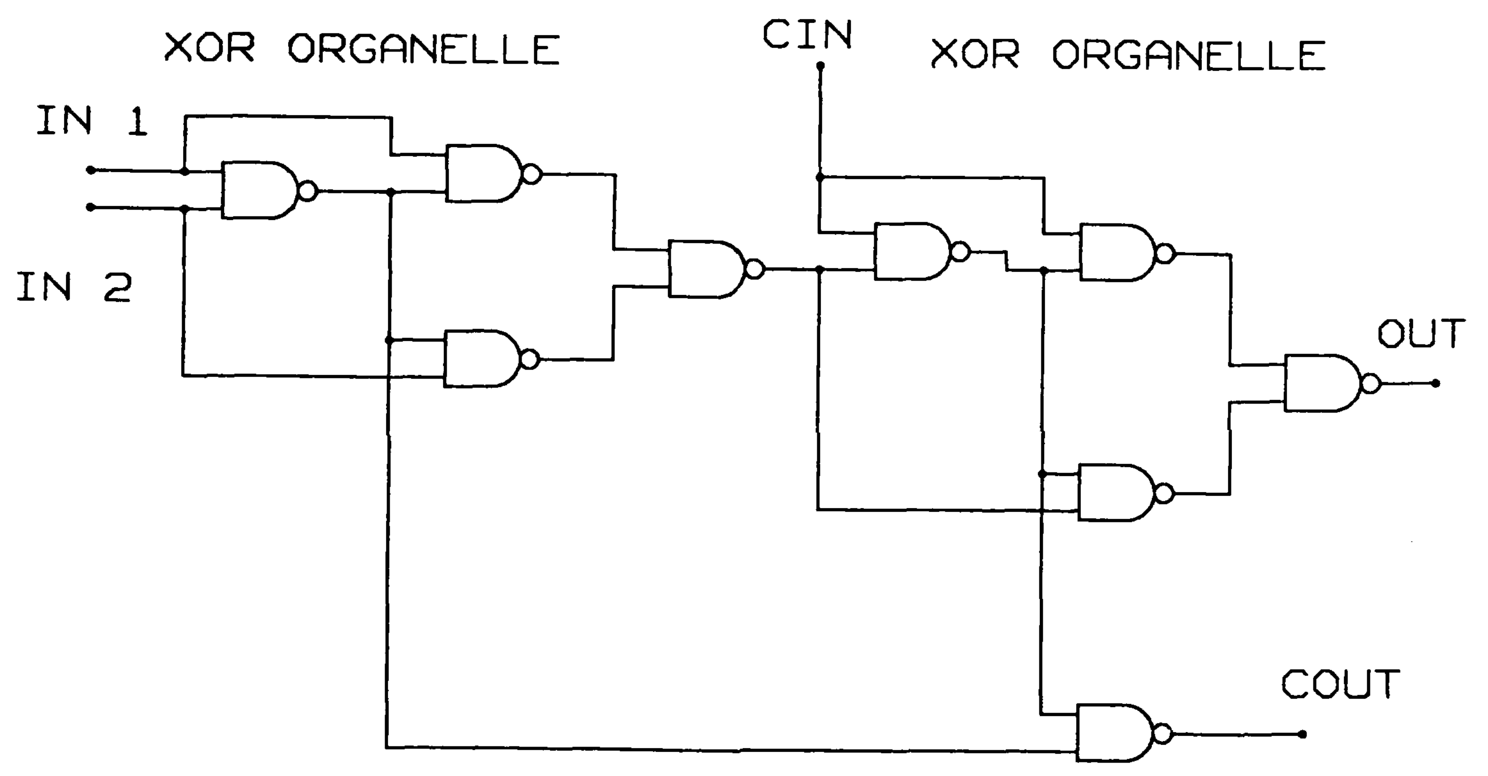


Figure 2.5. MacPitts Full Adder Design

The one-bit full adder circuit in Figure 2.5 shows how a complex function can be implemented by putting several different organelles together. In this case two XOR organelles and one NAND organelle are used. Figure 2.6 shows how the one-bit adder is used by MacPitts to build a two-bit full adder circuit with carry in.

Two different structures are used in the data-path to define and store organelle inputs and outputs. They are the internal port and the register, and both are the same size as the data word. Internal ports are used primarily to transfer the output of an organelle to another organelle or to the Weinberger array within the same clock cycle or state period [Ref. 2]. Registers are used to store word size data elements. A one-bit register organelle consists of a master-slave flip-flop, as shown in Figure 2.7, that is controlled by the MacPitts three-phase clock [Ref. 3]. This structure allows the output of the register to be valid during a clock cycle even though a new input value could be in the process of being clocked into the register. The enable line in Figure 2.7 is used to control which clock cycles the register samples the input line for data storage. A memory refresh cycle is performed if new data is not stored during a clock cycle. If data is to be stored in every clock cycle the enable line is connected to Vdd.

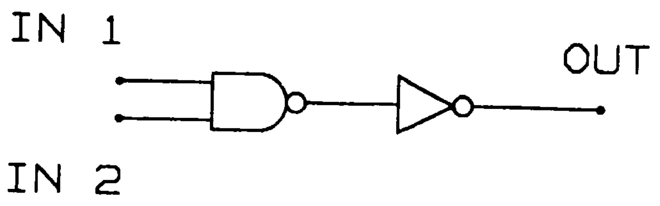


Figure 2.3. AND Organelle Structure

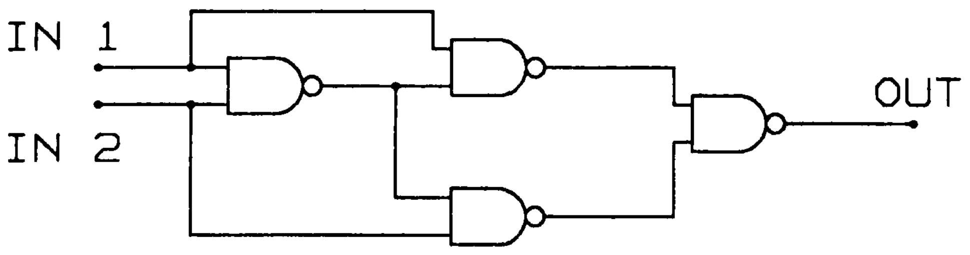


Figure 2.4. Exclusive-OR Organelle Structure

by the ordering of the word operations specified in the .mac program. The first word operation encountered by the compiler in the .mac program is the first organelle in the data-path and so on. The compiler takes into account the size requirements of each organelle to scale the amount of space between organelles to allow enough room for connection lines, control lines, power lines, and local interconnection buses. Power and ground buses are also sized based on organelle power requirements [Ref. 3].

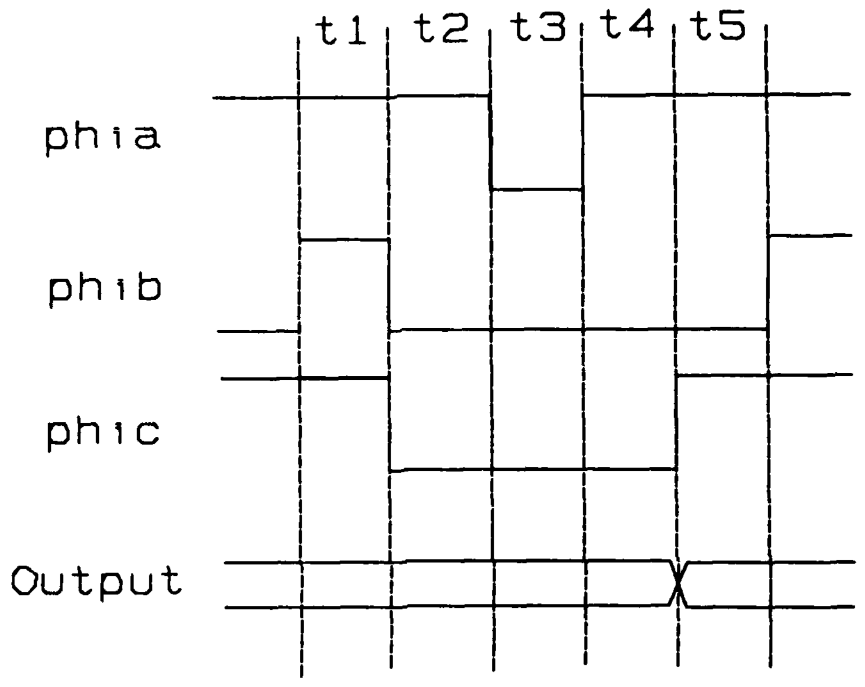
The routing of data to and from the data-path is very inefficient and requires many data lines to be longer than necessary. As seen in Figure 2.1, the chip pads are placed only on the top, right and bottom sides of the chip. Data entering the chip on input ports and exiting the chip through output ports is routed from the left side of the data-path. This causes very long data lines. Data from the data-path to the Weinberger array is routed from the bottom side of the data-path to the top side of the array.

All arithmetic and boolean function organelles are implemented using three basic gate structures. They are the NAND, NOR and inverter. Figure 2.3 shows an AND organelle that is made from a NAND gate and an inverter. In Figure 2.4 an XOR function is implemented using NAND gates. An OR gate is implemented from a NOR gate and an inverter and the boolean EQU function is implemented using four NOR gates.

C. THE DATA-PATH

The data-path is the unit where all word size operations are performed. These operations consist of arithmetic functions (addition, subtraction, incrementing, decrementing and equals), boolean functions (and, or, not, nand, nor, xor and equ), data shifting operations, comparison tests and data storage and transfer using registers and ports [Refs. 2 and 3]. The structure consisting of a one-bit slice of the above operations is referred to as an organelle and the LISP code used by the MacPitts compiler to generate each organelle can be found in the library and organelle sections of the MacPitts source code listing.

The size of the data-path is determined by the number of bits in the data word (specified at the beginning of the .mac program) and the number of word size operations to be performed. The number of bits in the data word specifies the height of the data-path. The larger the data word the taller the data-path. The width is determined by the number of functions performed. When a specific function is to be performed in the data-path the organelle that performs that function is placed in the data-path. Replicas of that organelle (one for each bit of the data word) are stacked on top of each other. The organelle for the most significant bit of the data word is on the bottom of the stack and the organelle for the least significant bit is on the top. The ordering of the organelles in the data-path is determined



t1 = static storage
t2 = isolate output
t3 = sample input
t4 = isolate input
t5 = connect to output

Figure 2.2. MacPitts Five Segment Clock Period

corner of the chip and continuing in a clockwise direction. The version of the MacPitts silicon compiler installed at the Naval Postgraduate School will not place pads on the left edge of the chip. A newer version of the compiler that is available commercially places pads on all four sides of the chip. All output pads are super buffered but the input data and clock pads are not.

Along with the ground and power pads, the three-phase clock pads must also be defined in all MacPitts programs even though the clock may not be used in the circuit. The clock bus is always laid out on the chip. The MacPitts compiler uses a three-phase overlapping clock scheme where the clock period is divided into five segments as shown in Figure 2.2. This unusual clock *scheme is* used to drive the data storage registers and flags (see paragraphs C and E below) and according to [Ref. 4] allows a more compact layout of the registers and flags.

A reset pad must also be defined if the "process" form is used in the .mac program even if the reset function is not used anywhere in the program. This is because the MacPitts compiler may use the reset signal in its internal algorithms when it generates the chip [Ref. 2]. If the "always" form is the only form used in the .mac program the reset pad is not required.

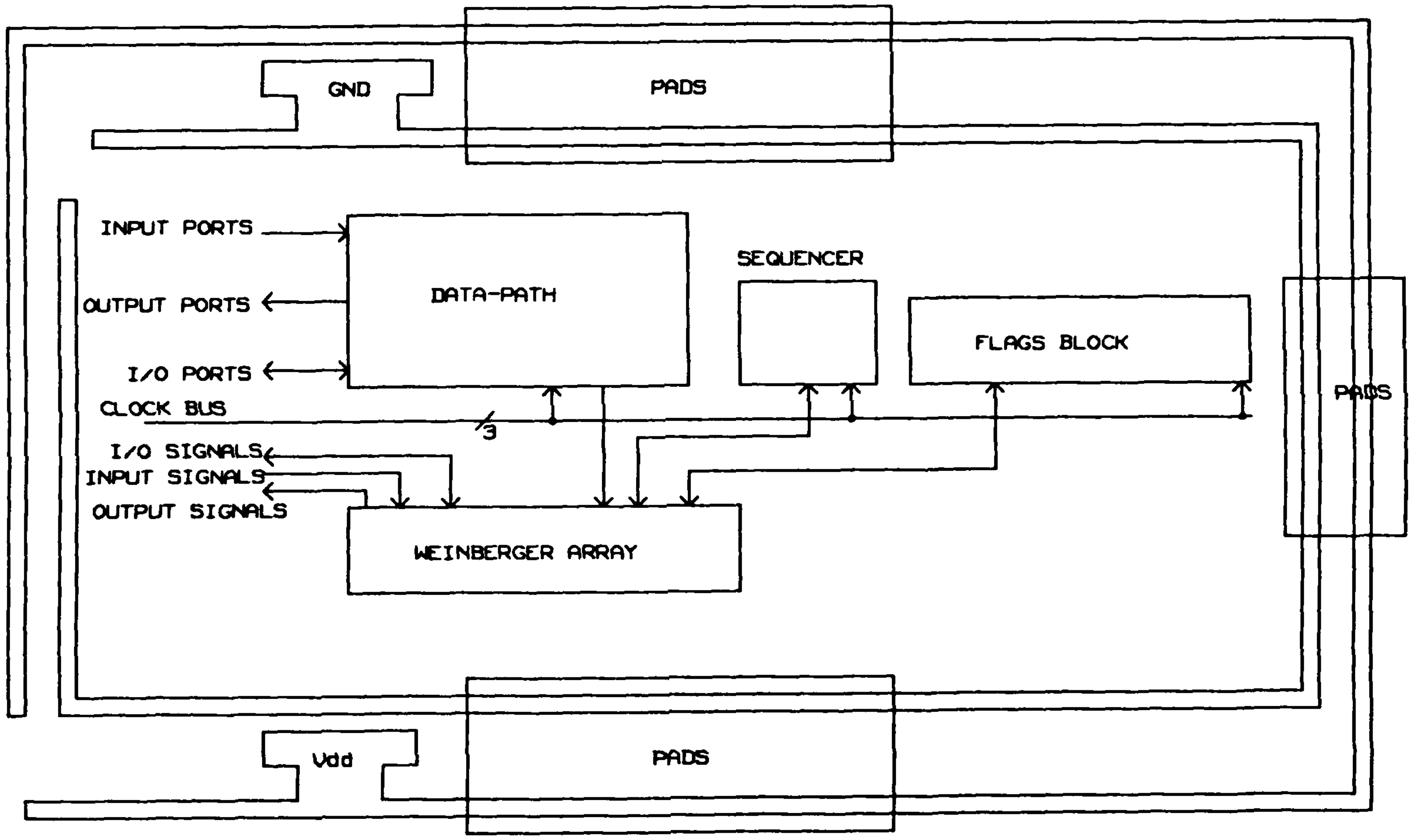


Figure 2.1. MacPitts Chip Design Structure

II. THE MACPITTS DESIGN ARCHITECTURE

A. INTRODUCTION

The MacPitts design structure consists of five main components. They are the chip design frame with pads, the data-path, the sequencer, the Weinberger array and the flags block (see Figure 2.1). Input ports or signals are used to bring input data into the chip and output ports or signals are used to output data from the chip. The difference between ports and signals is that a port has as many bits as the data word defined by the programmer in the MacPitts .mac program and a signal is only a one-bit data element.

B. THE DESIGN FRAME

The MacPitts compiler was designed to have no limit on the size of a circuit that it would design although large circuits may take several days of computer time to be completed. The design constraints that must be used for practical designs are the MOSIS chip size and pad number fabrication limitations. The current MOSIS limitation for the chip size is 7900 x 9200 microns and the maximum number of pads is 84.

All pads are defined in the "def" section of the MacPitts .mac program and are placed around the chip in the order specified in the program starting in the upper lefthand

Hauenstein adder [Ref. 5] along with a tutorial on Crystal. Design errors that have been found in MacPitts designs are detailed in Chapter V. Tutorial material on the MacPitts interpreter is found in Appendix A.

It was decided to use the MacPitts compiler to design various adder circuits so that performance (chip size, power and speed) comparisons could be made between the MacPitts designs and a hand-crafted pipeline adder circuit designed by Conradi and Hauenstein [Ref. 5]. Crystal, the VLSI timing analysis program developed at the University of California at Berkeley [Ref. 6], was used to analyze the timing requirements of all circuits being compared. Since Crystal had never been used at the Naval Postgraduate School before, a procedure on how to use Crystal to analyze MacPitts designs had to be developed. This required adapting the basic Crystal Commands to the unconventional MacPitts three-phase overlapping clock scheme.

The third research goal was to obtain a more complete understanding and description of the MacPitts interpreter than currently available in the literature. Reference 2 and reference 4 describe how to use the interpreter, but a detailed description of the interpreter commands and error statements and its capabilities and limitations is not available.

Chapter II of this thesis describes the basic circuit building blocks of the MacPitts compiler. The design of several combinational and pipeline adder circuits is presented in Chapter III. Chapter IV lists performance comparisons between the MacPitts adders and the Conradi and description of the MacPitts language, which is used to write the .mac program that contains a functional description of the circuit to be designed by the compiler. Also, a detailed procedure on how to write the .mac program is given. Carlson also shows how to use the MacPitts interpreter to test the functional correctness of the .mac program before the circuit design is performed by the compiler. In addition, Carlson's thesis gives a detailed listing of the activities in the MacPitts design cycle used to design VLSI circuits. The design cycle includes generating the .mac program, submitting the .mac program to the compiler for circuit design and performing a design rule check and functional event simulation on the designed circuit.

Since a good understanding of how to use the MacPitts silicon compiler to design VLSI circuits was obtained by Carlson [Ref. 4] it was decided that the next logical step was to learn more about the MacPitts architecture and to make some performance comparisons between various MacPitts and hand-crafted designs. The first goal of this thesis research was to determine what basic building blocks the compiler used to design VLSI circuits and how these building blocks are used to implement different circuits. Also, an understanding of how the statements in the .mac program determine the structure of the MacPitts designed circuit was desired.

D. THE SEQUENCER

The sequencer is a mini data-path and is placed on the chip between the data-path and the flags block. If the .mac program contains a process whose value depends on the system state, a sequencer is placed on the chip to control the system state of the chip. The sequencer usually contains registers to store the current system state. Every clock cycle the current system state is transferred to the Weinberger array from the registers and then the new system state is transferred from the Weinberger array to the sequencer for storage. Additional details about the sequencer are given in [Refs. 3, 7 and 8].

E. THE FLAGS BLOCK

Flags have a similar function to registers, but they store only one-bit of data from the Weinberger array. Flags also have a master-slave flip-flop structure but extra inverters are used in the flags block to drive the clock signals because there may be as many as twenty or thirty flags in the flags block (see Figure 2.8). The enable line of a flag performs the same function as the enable line of a register. Flags are placed side by side with the flags block increasing in width as more flags are needed. The rightmost structures in the flags block are the six inverters used to drive the three clock lines (see Figure 2.9). The leftmost flag in the flags block is the first flag encountered by the

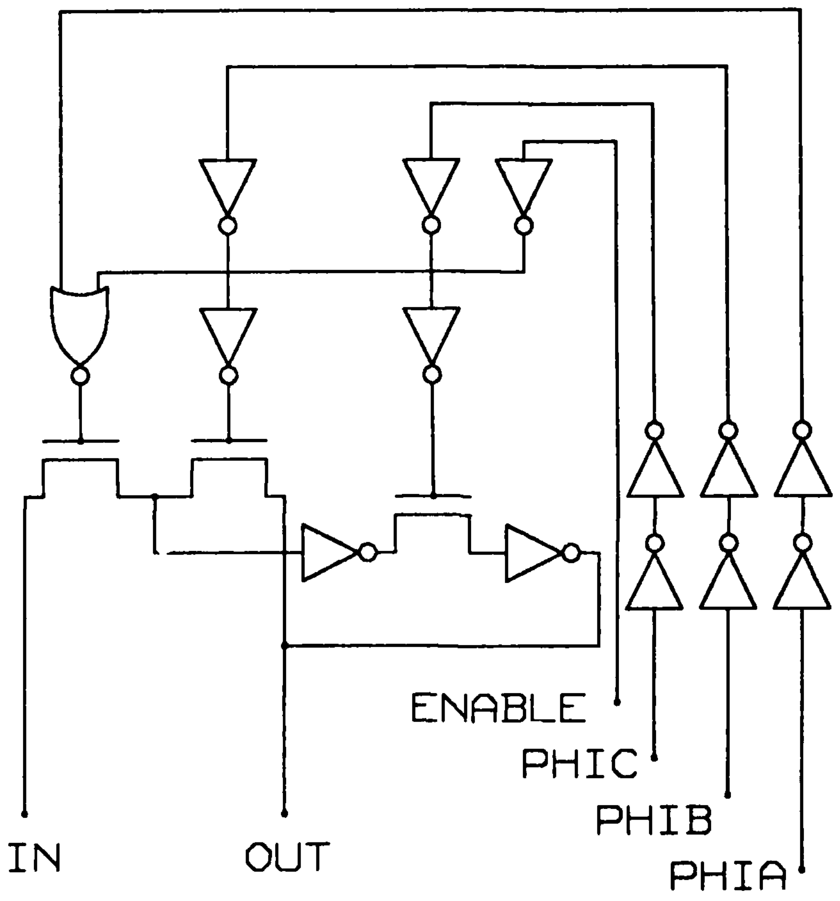


Figure 2.8. 1-Bit Flag Cell

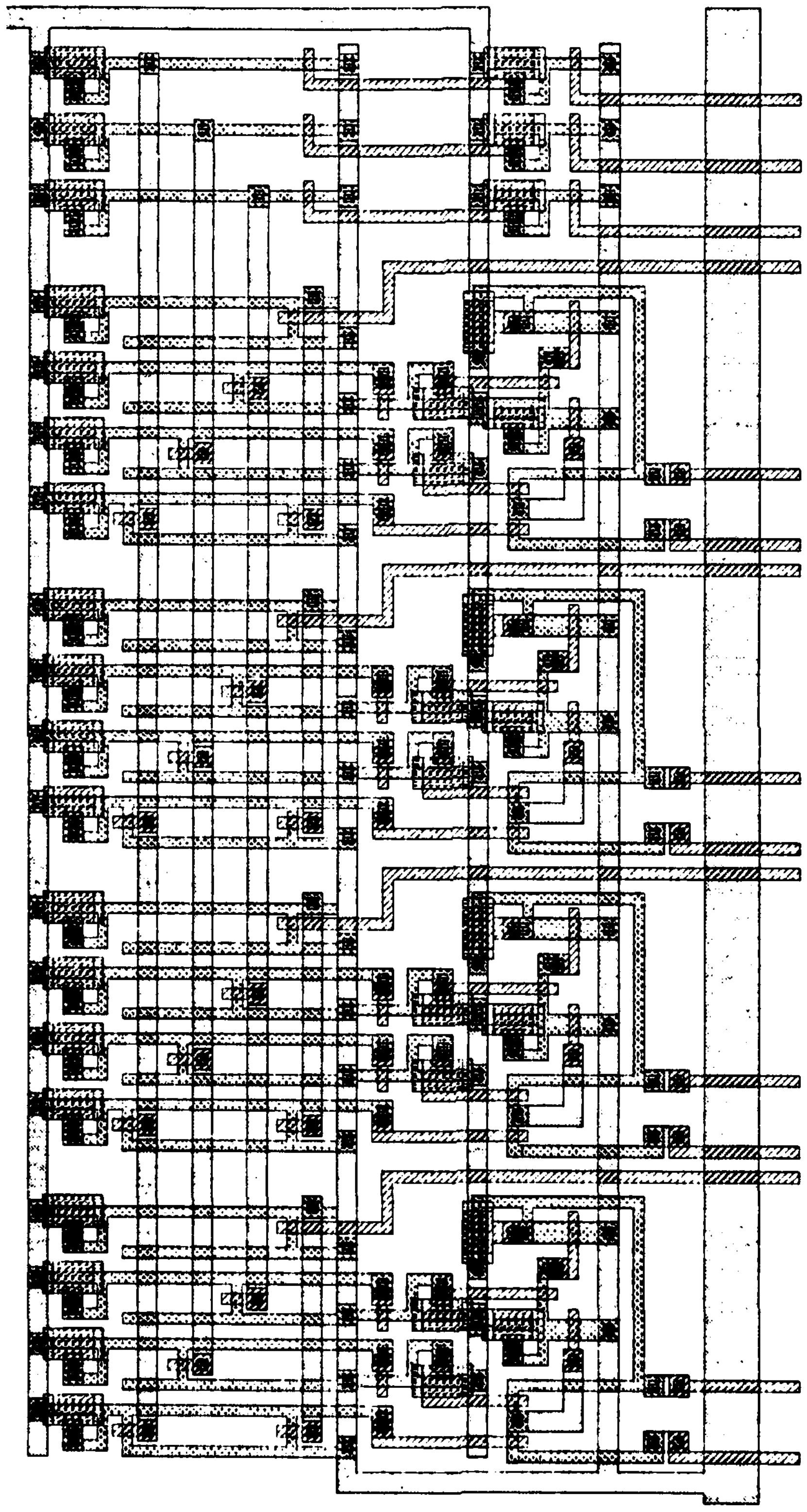


Figure 2.9. MacPitts Layout of a Four Cell Flags Block

compiler in the "always" or "process" section of the .mac program. Each subsequent flag encountered by the compiler is placed on the right of the previous flag. Since the flags block cannot expand in the vertical direction there is wasted space on the chip above the flags block if the data-path or sequencer is taller than the flags block. Also, if the .mac program requires a large number of flags the width of the flags block may make the dimension of the chip exceed the MOSIS chip size constraints.

F. THE WEINBERGER ARRAY

The Weinberger array, or control unit, or a MacPitts designed chip is the unit where all chip control signals are generated and bit size boolean functions are performed. All inputs and outputs to the array are routed to the top of the array. Input and output signal lines are routed around the left side of the array and then to the top.

The data lines connecting the Weinberger array to the data-path, sequencer, and flags block are called the "river". The algorithm that routes the "river" does not allow the data lines to cross each other so the left-to-right ordering of the functions performed in the array is determined by the left-to-right ordering of the data transferred from the data-path, sequencer and flags block to the array. Array functions that use data from the data-path are placed in the left section of the array, array functions that use data from the sequencer are placed in the center section of the array and array functions that use data from the flags block are placed in the right section of the array. Since no data lines in the "river" can cross each other data that is transferred between the data-path, sequencer or flags block must pass through the array even though no function is performed on the data in the array.

The Weinberger array consists of a regular structure of NOR gates having arbitrary numbers of inputs. The pull-up transistors of the NOR gates are connected to Vdd at the bottom of the array and run vertically the full height of the array. Vertical ground wires run parallel to the pull-up transistor lines from the ground bus at the top of the array. Inputs to the NOR gates run horizontally through the array and form pull-down transistors when connected to ground and the NOR gate output line. The NOR gate output lines also run horizontally through the array and may be used as input lines to other NOR gates or routed to a flag or signal output pad. As more NOR gates are added to the Weinberger array or more inputs or outputs are added to each gate the array increases in width. The height of the array is determined by the number of horizontal interconnections between the NOR gates [Ref. 7].

Eight different boolean functions are implemented in the Weinberger array, all with NOR gates: NOR, AND, NAND, OR, EQU, XOR, parity and NOT [Ref. 2]. Figure 2.10 shows how an

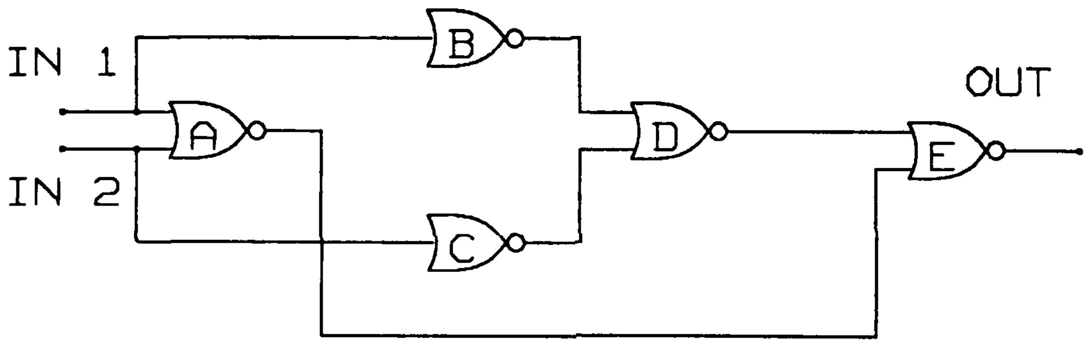


Figure 2.10. NOR Gate Implementation of the XOR Function

XOR function is implemented using NOR gates. The stick diagram of Figure 2.11 shows the Weinberger array implementation of the XOR function from Figure 2.10 and Figure 2.12 shows an actual Weinberger array layout of this function.

The PLA and Weinberger array structures are very similar but there are several important differences. First, the PLA has only two levels of logic, the AND and the OR planes. The Weinberger array can have an arbitrary NOR gate depth. Although a PLA can implement the same functions performed in the Weinberger array the MacPitts designers found that when a boolean function was normalized in the sum-of-products form the Weinberger array's NOR gate depth allowed a much more compact structure than the PLA's [Ref. 3]. Another difference is that the complement of each input signal does not have to be available at the input of the Weinberger array as a PLA requires. The complements of array inputs are generated in the array if they are required.

It has been found that the generation of the Weinberger array usually takes from 90% - 95% of the computer's compilation time in generating a MacPitts design. When an 8-bit 5-stage pipeline adder was designed using the MacPitts compiler 162 CPU minutes (about eight hours on a lightly loaded computer system) were required to complete this design. Most of this time was required to lay out the 228 vertical control columns (the number of array inputs and

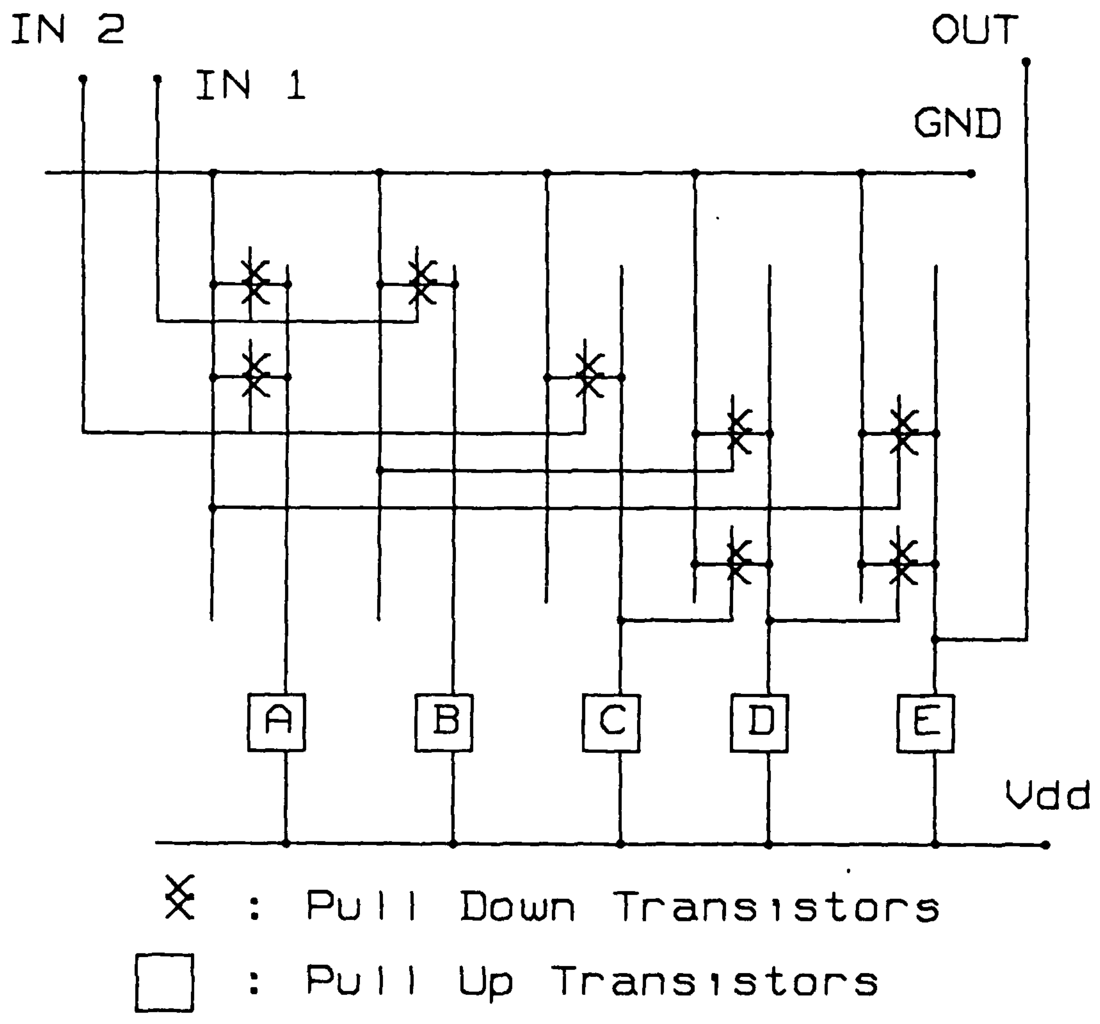


Figure 2.11. Stick Diagram of the Weinberger Array Implementation of an XOR Function

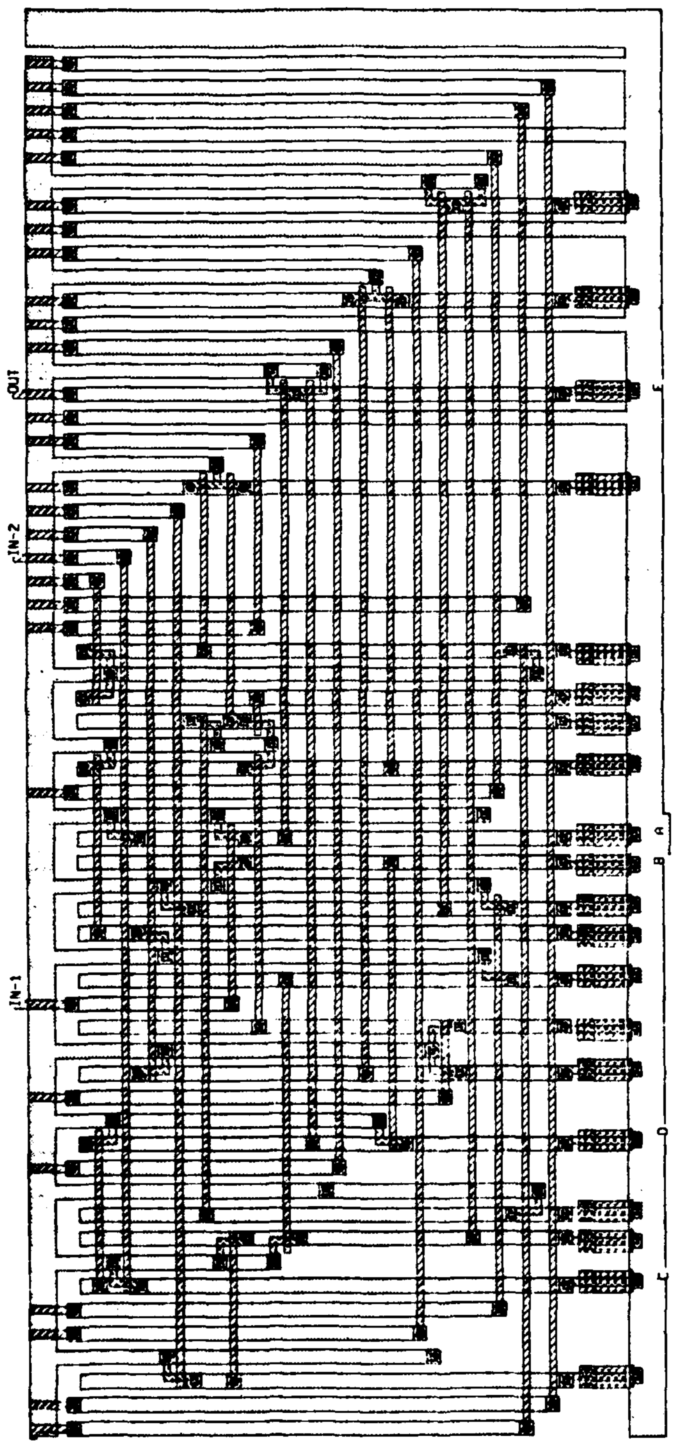


Figure 2.12. Weinberger Array Layout of an XOR Function

outputs plus the number of nor gates in the array) and the 81 horizontal control tracks (the number of nor gate inputs and outputs in the array). When a 16-bit 5-stage pipeline adder design was attempted which contained 435 control columns and 157 control tracks the design process was killed after 4800 CPU minutes (four days) were spent designing the Weinberger array. When the size of the Weinberger array of a 4-bit 5-stage pipeline adder (126 columsn and 43 tracks) is compared with the size of the 8-bit and 16-bit adders it can be seen that the Weinberger array becomes nearly four times larger and more complex in this 5-stage pipeline design when the size of the data word is doubled.

III. THE DESIGN OF ADDER CIRCUITS

A. COMBINATIONAL ADDERS

The design of combinational adder circuits with the MacPitts compiler is more straightforward than the design of pipeline adder circuits. The output sum of a combinational adder depends only on the present inputs to the circuit. Unfortunately, several compiler design constraints cause the combinational adder design to be more complicated than necessary.

The compiler adds two input vectors (ain and bin) in the data-path using the ripple carry full adder circuit shown in Figure 2.5. The first problem occurs when trying to add the input carry (cin) to the first bit of ain and bin. Since cin is a one-bit sized data element and the data-path can only manipulate word size data elements cin must be converted to a word sized data element. This requires additional circuitry in the data-path and the Weinberger array and also additional statements in the MacPitts .mac program.

A second problem occurs because the MacPitts language in which the .mac program is written allows only two variables in the addition function [Ref. 2]. All MacPitts functions are limited to one or two variables. It is assumed that the number of variables in a MacPitts function was limited by the compiler designers to simplify the design of the compiler. The simple LISP addition function of

```
(+ ain bin cin)
```

is accomplished in MacPitts with the more complicated function

```
(+ ain (+ bin cin)).
```

This embedded addition causes two full adder circuits to be connected in cascade. In the first full adder bin is added to cin and this sum is added to ain in the second full adder.

A third problem is that the carry in and carry out lines of the full adder cannot be addressed by the programmer. They are only used to ripple the carry bits between full adder stages. The carry in of the bit 0 full adder is connected to ground; the carry out of the last full adder stage is not connected to anything. If a chip carry out is desired it must be generated by additional circuitry in the Weinberger array.

Figure 3.1 shows a block diagram of the data-path for a two bit combinational adder circuit with carry in. In Figure 3.2 the .mac program for a 4-bit combinational adder is shown. Lines 14 and 15 convert the carry in signal to a word size data element. The least significant bit of the carry in word is set to 1 or 0 depending on the value of the carry in signal. All other bits of the carry in word are

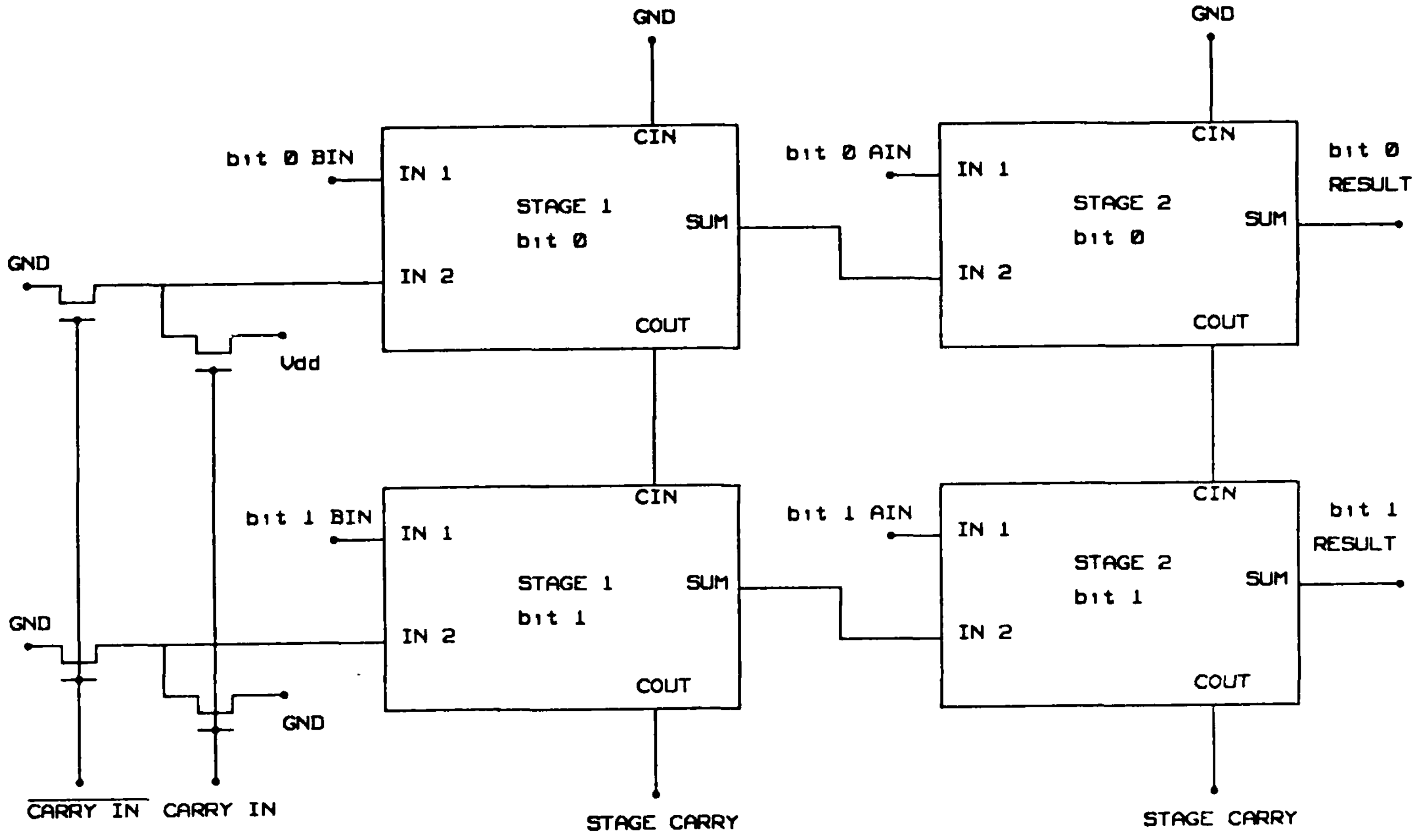


Figure 3.1. MacPitts Data-Path for a 2-Bit Full Adder Circuit

```
;adder 4-bit combinational
(program add 4
  (def 1 ground)
  (def ain port input (2 3 4 5))       ;input vector
  (def bin port input (6 7 8 9))       ;input vector
  (def res port output (10 11 12 13)) ;output vector
  (def cin signal input 14)           ;carry in
  (def carry port internal)
  (def 15 phia)
  (def 16 phib)
  (def 17 phic)
  (def 18 power)
  (always
    (cond (cin (setq carry 1))
          (t (setq carry 0)))
    (setq res (+ ain (+ bin carry)))))
```

Figure 3.2. 4-Bit Combinational Adder .mac Program

```
      (setq c5 (or (bit 5 g3) (and (bit 4 g3) (bit 5 p3))
          (and bc3 (bit 4 p3) (bit 5 p3))))
      (setq c6 (or (bit 6 g3) (and (bit 5 g3) (bit 6 p3))
          (and (bit 4 g3) (bit 5 p3) (bit 6 p3))
          (and bc3 (bit 4 p3) (bit 5 p3) (bit 6 p3))))
      (setq c7 bc7)
      (setq p4 p3)
      (setq carry4 carry3))
;
;Stage Five
;
  (par (setq add0 (xor (bit 0 p4) carry4))
       (setq add1 (xor (bit 1 p4) c0))
       (setq add2 (xor (bit 2 p4) c1))
       (setq add3 (xor (bit 3 p4) c2))
       (setq add4 (xor (bit 4 p4) c3))
       (setq add5 (xor (bit 5 p4) c4))
       (setq add6 (xor (bit 6 p4) c5))
       (setq add7 (xor (bit 7 p4) c6))
       (setq carryout c7)
       (setq sum0 add0)
       (setq sum1 add1)
       (setq sum2 add2)
       (setq sum3 add3)
       (setq sum4 add4)
       (setq sum5 add5)
       (setq sum6 add6)
       (setq sum7 add7)
       (setq cout carryout))))
```

Figure 3.8. MacPitts .mac Program for a 8-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
 (def add2 flag)
 (def add3 flag)
 (def add4 flag)
 (def add5 flag)
 (def add6 flag)
 (def add7 flag)
 (always
;
;Stage One
;
  (par (setq p1 (word-xor ain bin))
       (setq g1 (word-and ain bin))
       (cond (cin (setq carry1 t))
             (t (setq carry1 f))))
;
;Stage Two
;
  (par (setq bp0 (and (bit 3 p1) (bit 2 p1) (bit 1 p1) (bit 0 p1)))
       (setq bp1 (and (bit 7 p1) (bit 6 p1) (bit 5 p1) (bit 4 p1)))
       (setq bg0 (or (bit 3 g1) (and (bit 2 g1) (bit 3 p1))
           (and (bit 1 g1) (bit 2 p1) (bit 3 p1))
           (and (bit 0 g1) (bit 1 p1) (bit 2 p1) (bit 3 p1))))
       (setq bg1 (or (bit 7 g1) (and (bit 6 g1) (bit 7 p1))
           (and (bit 5 g1) (bit 6 p1) (bit 7 p1))
           (and (bit 4 g1) (bit 5 p1) (bit 6 p1) (bit 7 p1))))
       (setq p2 p1)
       (setq g2 g1)
       (setq carry2 carry1))
;
;Stage Three
;
  (par (setq bc3 (or bg0 (and carry2 bp0)))
       (setq bc7 (or bg1 (and bg0 bp1) (and carry2 bp0 bp1)))
       (setq p3 p2)
       (setq g3 g2)
       (setq carry3 carry2))
;
;Stage Four
;
  (par (setq c0 (or (bit 0 g3) (and carry3 (bit 0 p3))))
       (setq c1 (or (bit 1 g3) (and (bit 0 g3) (bit 1 p3))
           (and carry3 (bit 0 p3) (bit 1 p3))))
       (setq c2 (or (bit 2 g3) (and (bit 1 g3) (bit 2 p3))
           (and carry3 (bit 0 p3) (bit 1 p3) (bit 2 p3))))
       (setq c3 bc3)
       (setq c4 (or (bit 4 g3) (and bc3 (bit 4 g3))))
```

Figure 3.8. MacPitts .mac Program for a 8-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
(program addp 8
;This adder uses block carry lookahead (BCLA) addition
 (def 1 ground)
 (def ain port input (2 3 4 5 6 7 8 9)) ;input vector
 (def bin port input (10 11 12 13 14 15 16 17)) ;input vector
 (def cin signal input 18)        ;carry into chip
 (def sum7 signal output 19)    ;bit 7 sum
 (def sum6 signal output 20)    ;bit 6 sum
 (def sum5 signal output 21)    ;bit 5 sum
 (def sum4 signal output 22)    ;bit 4 sum
 (def sum3 signal output 23)    ;bit 3 sum
 (def sum2 signal output 24)    ;bit 2 sum
 (def sum1 signal output 25)    ;bit 1 sum
 (def sum0 signal output 26)    ;bit 0 sum
 (def cout signal output 27)    ;carry out of chip
 (def 28 phia)                 ;clock phases
 (def 29 phib)
 (def 30 phic)
 (def 31 power)
 (def p1 register)             ;carry propagate-stage one
 (def p2 register)             ;               -stage two
 (def p3 register)             ;               -stage three
 (def p4 register)             ;               -stage four
 (def g1 register)             ;carry generate-stage one
 (def g2 register)             ;              -stage two
 (def g3 register)             ;              -stage three
 (def bp0 flag)                ;block carry propagate
 (def bp1 flag)
 (def bg0 flag)                ;block carry generate
 (def bg1 flag)
 (def bc3 flag)                ;block carry
 (def bc7 flag)
 (def carry1 flag)             ;cin-stage one
 (def carry2 flag)             ;   -stage two
 (def carry3 flag)             ;   -stage three
 (def carry4 flag)             ;   -stage four
 (def c0 flag)                 ;bit 0 carry
 (def c1 flag)                 ;bit 1 carry
 (def c2 flag)                 ;bit 2 carry
 (def c3 flag)                 ;bit 3 carry
 (def c4 flag)                 ;bit 4 carry
 (def c5 flag)                 ;bit 5 carry
 (def c6 flag)                 ;bit 6 carry
 (def c7 flag)                 ;bit 7 carry
 (def carryout flag)           ;cout flag
 (def add0 flag)               ;bit sum flags
 (def add1 flag)
```

Figure 3.8. MacPitts .mac Program for a 8-Bit 5-Stage Pipeline Adder Circuit

8-bit adder is shown in Figure 3.8 and the circuit layout is shown in Figure 3.9. The block diagram of the 8-bit adder would be the same as the block diagram of the 4-bit adder shown in Figure 3.7. The size of the 8-bit adder circuit is 6.650mm x 4.358mm. The data-path is twice as tall, the flags block is almost twice as long and the area of the Weinberger array is four times larger in the 8-bit adder than in the 4-bit adder.

An attempt was made to design a 16-bit 5-stage pipeline adder with the MacPitts compiler. The compiler was able to design all but the large Weinberger array which is four times larger than the 8-bit adder array. The program that designs the Weinberger array uses a recursive algorithm and the depth of recursion is limited by the amount of memory available to the LISP compiler. Since the array of the 16-bit adder circuit is so large the limit of the depth of recursion was reached.

The 16-bit pipeline adder contains four carry-look-ahead blocks. When the .mac program of the 16-bit adder (Figure 3.10) is compared to the .mac programs of the 8-bit and 4-bit adders (Figures 3.6 and 3.9) the programs are essentially the same except for additional statements in stages 2 through 5 due to the larger 16-bit data word and due to the additional carry-look-ahead blocks.

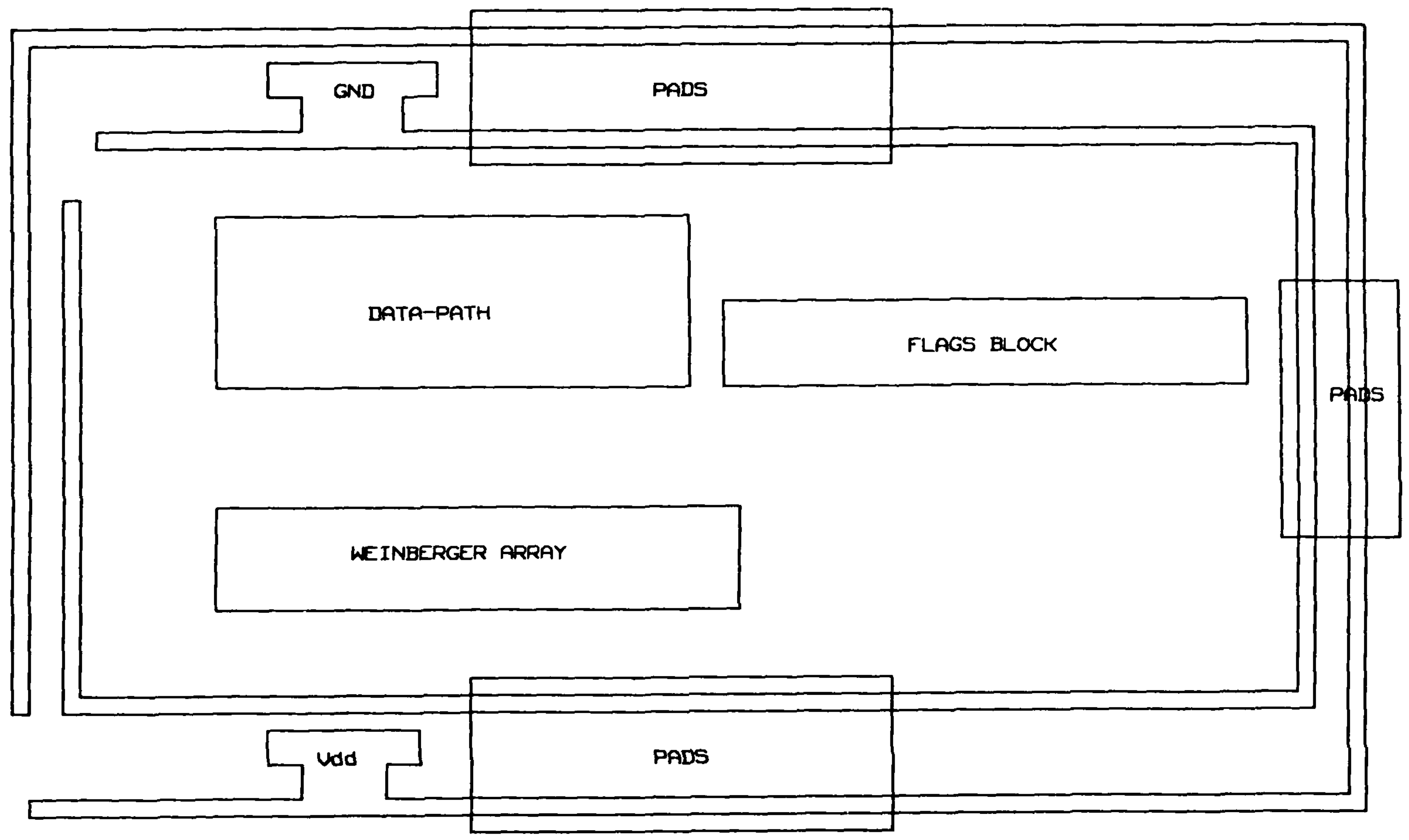


Figure 3.7. Block Diagram of a 4-Bit 5-Stage Pipeline Adder Circuit

shown in lines 43 and 44, are performed with the word-xor and the word-and functions in the data-path. All functions in stages 2 thru 5 require the manipulation of bit size data elements. These functions are performed in a large Weinberger array. Registers and flags are used to store the input and output data of each stage. It takes less circuitry (and fewer statements in the .mac program) to manipulate word size data elements and store them in registers than to manipulate bit size data elements and store them in flags. Since there is no MacPitts function to set the bits of a word to a particular value the bit sized output data elements of stages 2 through 4 cannot be combined into words and stored in registers. The output data of stages 2 through 4 must be stored in flags and this requires a very large flags block.

A pipeline circuit designed by the MacPitts compiler does not perform like a standard pipeline circuit as described in [Ref. 9] because the input data of each stage is valid before the start of the clock period. When data is stored in a MacPitts register or flag the data is valid on the register or flag output line before the end of the clock period (see Figures 2.2, 2.7, and 2.8). The data then starts to propagate through the combinational logic of the next stage before the start of the next clock period. During the next clock period the data will continue to propagate through the stage combinational logic during the first two

```
   (par (setq p1 (word-xor ain bin))
        (setq g1 (word-and ain bin))
        (cond (cin (setq carry1 t))
              (t (setq carry1 f))))
;
;Stage Two
;
   (par (setq bp0 (and (bit 3 p1) (bit 2 p1) (bit 1 p1) (bit 0 p1)))
        (setq bg0 (or (bit 3 g1) (and (bit 2 g1) (bit 3 p1))
            (and (bit 1 g1) (bit 2 p1) (bit 3 p1))
            (and (bit 0 g1) (bit 1 p1) (bit 2 p1) (bit 3 p1))))
        (setq p2 p1)
        (setq g2 g1)
        (setq carry2 carry1))
;
;Stage Three
;
   (par (setq bc3 (or bg0 (and carry2 bp0)))
        (setq p3 p2)
        (setq g3 g2)
        (setq carry3 carry2))
;
;Stage Four
;
   (par (setq c0 (or (bit 0 g3) (and carry3 (bit 0 p3))))
        (setq c1 (or (bit 1 g3) (and (bit 0 g3) (bit 1 p3))
            (and carry3 (bit 0 p3) (bit 1 p3))))
        (setq c2 (or (bit 2 g3) (and (bit 1 g3) (bit 2 p3))
            (and carry3 (bit 0 p3) (bit 1 p3) (bit 2 p3))))
        (setq p4 p3)
        (setq c3 bc3)
        (setq carry4 carry3))
;
;Stage Five
;
   (par (setq add0 (xor (bit 0 p4) carry4))
        (setq add1 (xor (bit 1 p4) c0))
        (setq add2 (xor (bit 2 p4) c1))
        (setq add3 (xor (bit 3 p4) c2))
        (setq carryout c3)
        (setq sum0 add0)
        (setq sum1 add1)
        (setq sum2 add2)
        (setq sum3 add3)
        (setq cout carryout))))
```

Figure 3.5. MacPitts .mac Program for a 4-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
(program addp 4
,This adder uses block carry lookahead (BCLA) addition
 (def 1 ground)
 (def ain port input (2 3 4 5)) ;input vector
 (def bin port input (6 7 8 9)) ;input vector
 (def cin signal input 10)       ;carry into chip
 (def sum3 signal output 11)     ;bit 3 sum
 (def sum2 signal output 12)     ;bit 2 sum
 (def sum1 signal output 13)     ;bit 1 sum
 (def sum0 signal output 14)     ;bit 0 sum
 (def cout signal output 15)     ;carry out of chip
 (def 16 phia)                   ;clock phases
 (def 17 phib)
 (def 18 phic)
 (def 19 power)
 (def p1 register)               ;carry propagate-stage one
 (def p2 register)               ;               -stage two
 (def p3 register)               ;               -stage three
 (def p4 register)               ;               -stage four
 (def g1 register)               ;carry generate-stage one
 (def g2 register)               ;              -stage two
 (def g3 register)               ;              -stage three
 (def bp0 flag)                  ;block carry propagate
 (def bg0 flag)                  ;block carry generate
 (def bc3 flag)                  ;block carry
 (def carry1 flag)               ;cin-stage one
 (def carry2 flag)               ;   -stage two
 (def carry3 flag)               ;   -stage three
 (def carry4 flag)               ;   -stage four
 (def c0 flag)                   ;bit 0 carry
 (def c1 flag)                   ;bit 1 carry
 (def c2 flag)                   ;bit 2 carry
 (def c3 flag)                   ;bit 3 carry
 (def carryout flag)             ;cout flag
 (def add0 flag)                 ;bit sum flags
 (def add1 flag)
 (def add2 flag)
 (def add3 flag)
 (always
;
;Stage One
;
```

Figure 3.5. MacPitts .mac Program for a 4-Bit 5-Stage Pipeline Adder Circuit

$$C_{2,6,10,14}=G_{2,6,10,14}+G_{1,5,9,13}P_{2,6,10,14}$$
$$+G_{0,4,8,12}P_{1,5,9,13}P_{2,6,10,14}$$
$$+BC_{-1,3,7,11}P_{0,4,8,12}P_{1,5,9,13}P_{2,6,10,14}$$

(Note that $C_{0,4,8,12}$ means C_i for i=0,4,8,12.)

5. Calculate the sum bits (S_i).

$$S_i = (A_i)XOR(B_i)XOR(C_{i-1})$$

The Conradi and Hauenstein [Ref. 5] pipeline adder had only four stages. Stages 1 and 2 were combined by writing the equations describing the BG_j's and BP_j's in terms of the input operands instead of in terms of the G_i's and P_i's. The MacPitts pipeline adders contain five stages because the increased stage propagation delay caused by combining stages 1 and 2 could slow the clock speed of the circuit and the fastest possible clock speed is desired.

Figure 3.5 shows the .mac program for a 4-bit 5-stage pipeline adder circuit. The carry in of the chip is used in all stages of the pipeline so a separate storage location is required for each stage as shown in lines 26 thru 29. The carry propagate and carry generate calculated in stage 1 are used in stages 4 and 5, respectively, so multiple storage locations are also used for these quantities. The calculations of the carry generate and carry propagate,

2. Calculate one block generate (BG_j) and block propagate (BP_j) for every four bits of the addition operands from the G_i's and P_i's.

$$BP_j = P_{i+3}P_{i+2}{}^{+}{}_{i+1}P_i \text{ ; } j=0,1,2,3,\dots \text{ ; } i=4j$$

$$BG_j = G_{i+3}+G_{i+2}+P_{i+3}+G_{i+1}P_{i+2}P_{i+3}+G_iP_{i+1}P_{i+2}P_{i+3} \text{ ; }$$
$$j=0,1,2,3,\dots \text{ ; } i=4j$$

3. Calculate the block carry (BC_l) for each carry block.

$$BC_3 = BG_0+C_{-1}BP_0$$

$$BC_7 = BG_1+BG_0BP_1+C_{-1}BP_0BP_1$$

$$BC_{11} = BG_2+BG_1BP_2+BG_0BP_1BP_2+C_{-1}BP_0BP_1BP_2$$

$$BC_{15} = BG_3+BG_2BP_3+BG_1BP_2BP_3+BG_0BP_1BP_2BP_3+C_{-1}BP_0BP_1BP_2BP_3$$

4. Calculate the look-ahead-carry (C_i) for each bit of the operands.

$$C_{0,4,8,12}=G_{0,4,8,12}+BC_{-1,3,7,11}P_{0,4,8,12}$$

$$C_{1,5,9,13}=G_{1,5,9,13}+G_{0,4,8,12}P_{1,5,9,13}$$
$$+BC_{-1,3,7,11}P_{0,4,8,12}P_{1,5,9,13}$$

combinational logic of the combinational circuit the pipeline circuit has a shorter logic propagation delay during each clock period. This allows the pipeline circuit to operate at a faster clock speed and higher data output rate (throughput) than the combinational circuit. A disadvantage of a pipeline circuit is the latency caused by the time that is required to fill and empty the pipeline. Reference 9 should be consulted for more information on pipelining.

There are many different algorithms that can be used to design a pipeline adder circuit. The block carry-look-ahead (BLCA) addition algorithm [Ref. 10] was used so that a comparison could be made between a MacPitts designed pipeline adder circuit and the hand-crafted pipeline adder circuit designed by Conradi and Hauenstein [Ref. 5]. Equations 6.1 thru 6.12 of [Ref. 5] are used to implement the BCLA addition algorithm. As described in [Ref. 5], the BLCA pipeline adder can be conveniently divided into the following five stages:

1. Calculate the carry generate (G_i) and the carry propagate (P_i) from the input addition operands.

   $G_i = A_iB_i$

   $P_i = (A_i)XOR(B_i)$

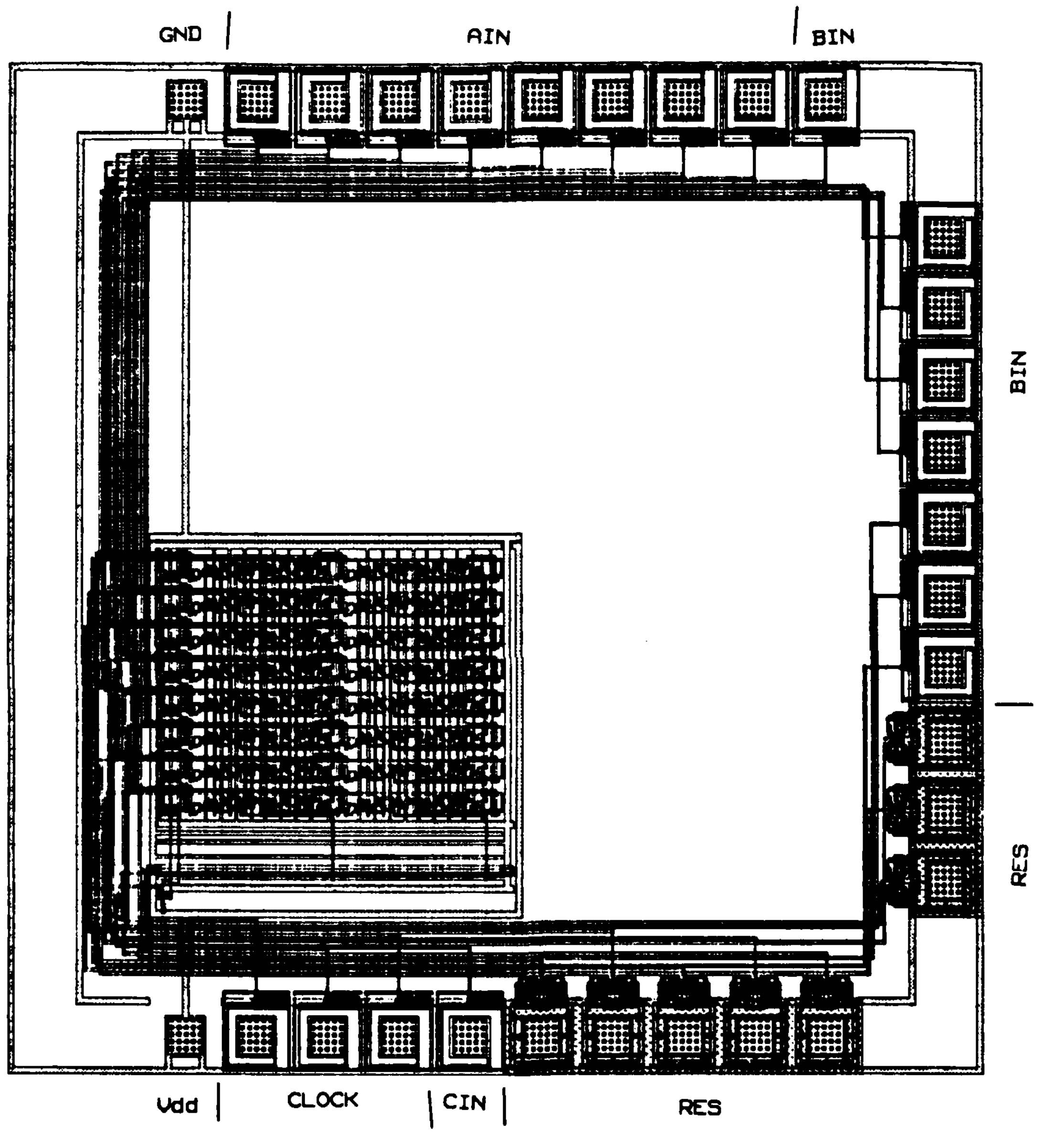


Figure 3.4. MacPitts Design of a 8-Bit Combinational Adder

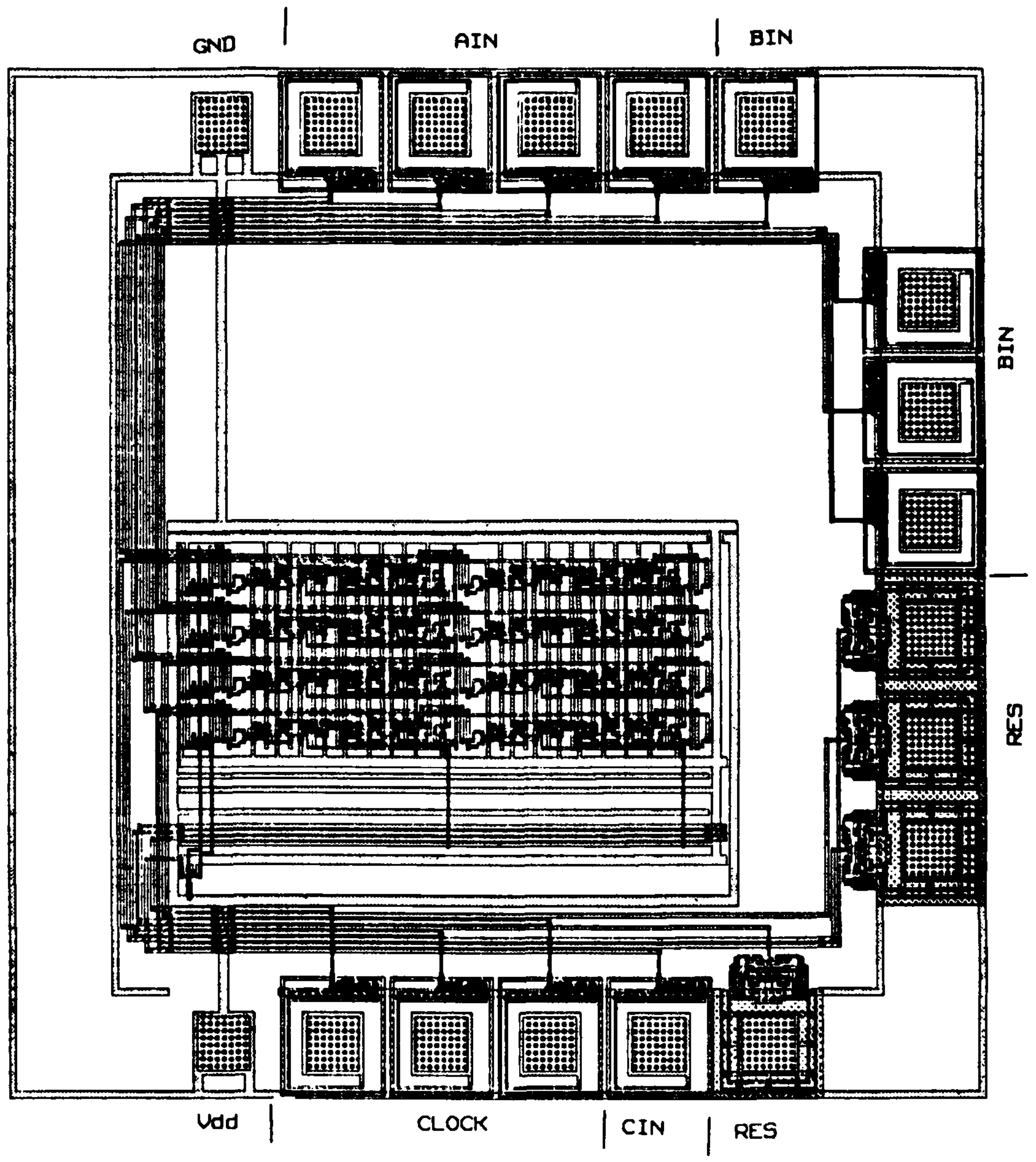


Figure 3.3. MacPitts Design of a 4-Bit Combinational Adder

set to 0 regardless of the value of the carry in signal. This can be seen on the left side of Figure 3.1. Figure 3.3 and 3.4 show the 4 micron MacPitts designs for a 4-bit and an 8-bit combinational adder. The size of the 4-bit adder is 2.292mm x 2.398mm and the size of the 8-bit adder is 3.508mm x 3.614mm. As shown, the size of the chip frame is larger than required by the circuitry inside the chip. A larger frame is needed because pads can be placed only on three sides of the frame. The frame could be smaller and the chip area could be more effectively used if pads were placed on all four sides of the frame. Both of these MacPitts designs produced correct simulations when simulated by the event driven switch level simulator, esim, using the procedure outlined in [Ref. 4].

B. PIPELINE ADDERS

The purpose of pipelining a circuit is to increase the throughput of the circuit. The combinational logic of a circuit is partitioned into several smaller functional units or stages and storage registers are placed between each stage. During each clock period data is clocked from the input storage register of each stage through the combinational logic of the stage and into the output storage register of the stage. Also, during each clock period a result exits the pipeline. Since the combinational logic in each stage of the pipeline circuit is less than the total

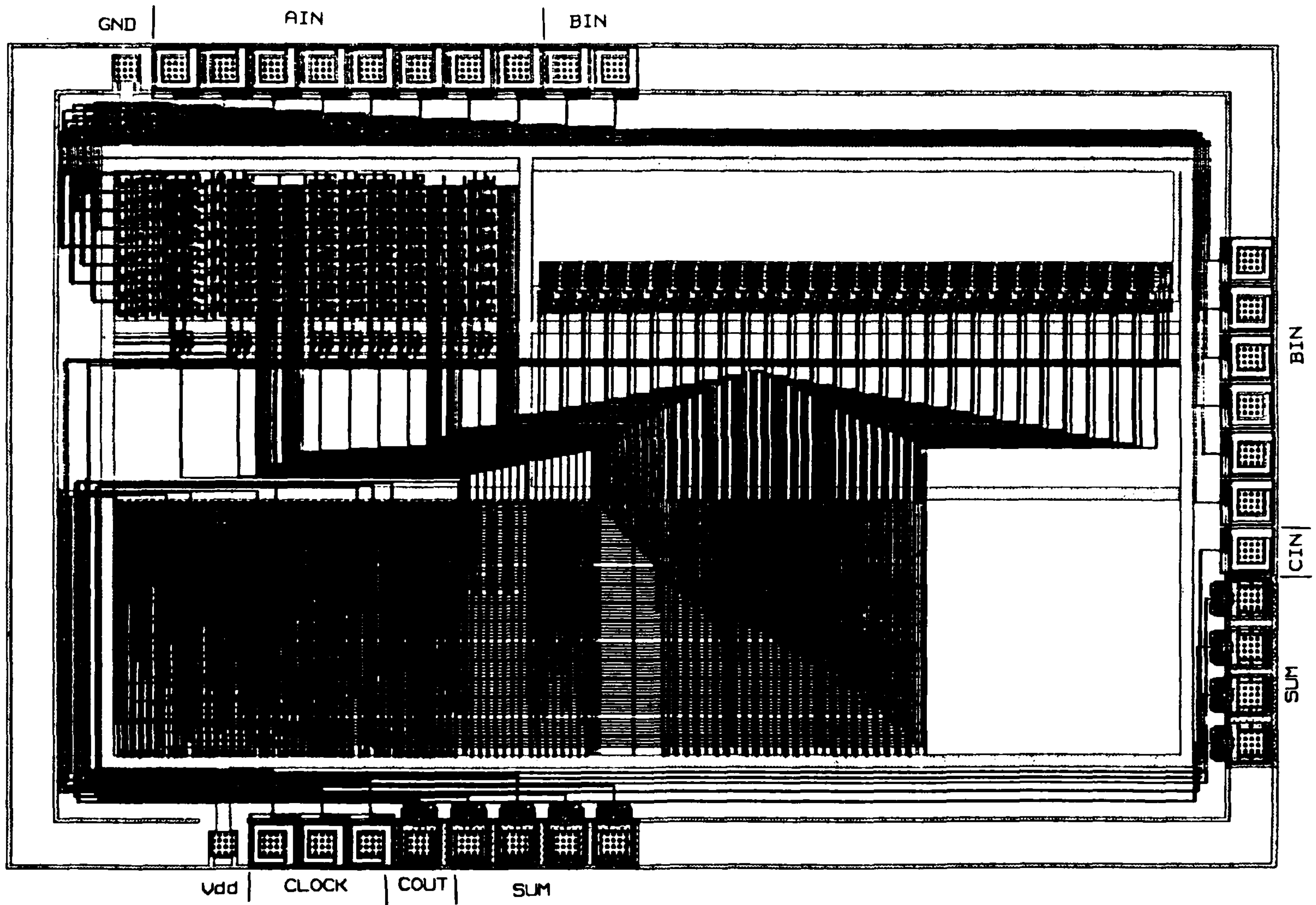


Figure 3.9. Layout of a 8-Bit 5-Stage Adder Circuit

```
(program addp 16
;This adder uses block carry lookahead (BCLA) addition
  (def 1 ground)
  (def ain port input (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17))
  (def bin port input (18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33))
  (def cin signal input 34)
  (def sum15 signal output 35)
  (def sum14 signal output 36)
  (def sum13 signal output 37)
  (def sum12 signal output 38)
  (def sum11 signal output 39)
  (def sum10 signal output 40)
  (def sum9 signal output 41)
  (def sum8 signal output 42)
  (def sum7 signal output 43)
  (def sum6 signal output 44)
  (def sum5 signal output 45)
  (def sum4 signal output 46)
  (def sum3 signal output 47)
  (def sum2 signal output 48)
  (def sum1 signal output 49)
  (def sum0 signal output 50)
  (def cout signal output 51)
  (def 52 phia)
  (def 53 phib)
  (def 54 phic)
  (def 55 power)
  (def p1 register)
  (def p2 register)
  (def p3 register)
  (def p4 register)
  (def g1 register)
  (def g2 register)
  (def g3 register)
  (def bp0 flag)
  (def bp1 flag)
  (def bp2 flag)
  (def bp3 flag)
  (def bg0 flag)
  (def bg1 flag)
```

Figure 3.10. MacPitts .mac Program for a 16-Bit 5-Stage Pipeline Adder Circuit

```
(def bg2 flag)
(def bg3 flag)
(def bc3 flag)
(def bc7 flag)
(def bc11 flag)
(def bc15 flag)
(def carry1 flag)
(def carry2 flag)
(def carry3 flag)
(def carry4 flag)
(def c0 flag)
(def c1 flag)
(def c2 flag)
(def c3 flag)
(def c4 flag)
(def c5 flag)
(def c6 flag)
(def c7 flag)
(def c8 flag)
(def c9 flag)
(def c10 flag)
(def c11 flag)
(def c12 flag)
(def c13 flag)
(def c14 flag)
(def c15 flag)
(def carryout flag)
(def add0 flag)
(def add1 flag)
(def add2 flag)
(def add3 flag)
(def add4 flag)
(def add5 flag)
(def add6 flag)
(def add7 flag)
(def add8 flag)
(def add9 flag)
(def add10 flag)
(def add11 flag)
(def add12 flag)
```

Figure 3.10. MacPitts .mac Program for a 16-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
(def add13 flag)
(def add14 flag)
(def add15 flag)
(always
;
;Stage One
;
  (par (setq p1 (word-xor ain bin))
       (setq g1 (word-and ain bin))
       (cond (cin (setq carry1 t))
             (t (setq carry1 f))))
;
;Stage Two
;
  (par (setq bp0 (and (bit 3 p1) (bit 2 p1) (bit 1 p1) (bit 0 p1)))
       (setq bp1 (and (bit 7 p1) (bit 6 p1) (bit 5 p1) (bit 4 p1)))
       (setq bp2 (and (bit 11 p1) (bit 10 p1) (bit 9 p1) (bit 8 p1)))
       (setq bp3 (and (bit 15 p1) (bit 14 p1) (bit 13 p1)
                 (bit 12 p1)))
       (setq bg0 (or (bit 3 g1) (and (bit 2 g1) (bit 3 p1))
                 (and (bit 1 g1) (bit 2 p1) (bit 3 p1))
                 (and (bit 0 g1) (bit 1 p1) (bit 2 p1) (bit 3 p1))))
       (setq bg1 (or (bit 7 g1) (and (bit 6 g1) (bit 7 p1))
                 (and (bit 5 g1) (bit 6 p1) (bit 7 p1))
                 (and (bit 4 g1) (bit 5 p1) (bit 6 p1) (bit 7 p1))))
       (setq bg2 (or (bit 11 g1) (and (bit 10 g1) (bit 11 p1))
                 (and (bit 9 g1) (bit 10 p1) (bit 11 p1))
                 (and (bit 8 g1) (bit 9 p1) (bit 10 p1) (bit 11 p1))))
       (setq bg3 (or (bit 15 g1) (and (bit 14 g1) (bit 15 p1))
                 (and (bit 13 g1) (bit 14 p1) (bit 15 p1))
                 (and (bit 12 g1) (bit 13 p1) (bit 14 p1) (bit 15 p1))))
       (setq p2 p1)
       (setq g2 g1)
       (setq carry2 carry1))
;
;Stage Three
;
  (par (setq bc3 (or bg0 (and carry2 bp0)))
       (setq bc7 (or bg1 (and bg0 bp1) (and carry2 bp0 bp1)))
       (setq bc11 (or bg2 (and bg1 bp2) (and bg0 bp1 bp2)
```

Figure 3.10. MacPitts .mac Program for a 16-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
              (and carry2 bp0 bp1 bp2)))
        (setq bc15 (or bg3 (and bg2 bp3) (and bg1 bp2 bp3)
              (and bg0 bp1 bp2 bp3) (and carry2 bp0 bp1 bp2 bp3)))
        (setq p3 p2)
        (setq g3 g2)
        (setq carry3 carry2))
;
;Stage Four
;
   (par (setq c0 (or (bit 0 g3) (and carry3 (bit 0 p3))))
        (setq c1 (or (bit 1 g3) (and (bit 0 g3) (bit 1 p3))
              (and carry3 (bit 0 p3) (bit 1 p3))))
        (setq c2 (or (bit 2 g3) (and (bit 1 g3) (bit 2 p3))
              (and carry3 (bit 0 p3) (bit 1 p3) (bit 2 p3))))
        (setq c3 bc3)
        (setq c4 (or (bit 4 g3) (and bc3 (bit 4 g3))))
        (setq c5 (or (bit 5 g3) (and (bit 4 g3) (bit 5 p3))
              (and bc3 (bit 4 p3) (bit 5 p3))))
        (setq c6 (or (bit 6 g3) (and (bit 5 g3) (bit 6 p3))
              (and (bit 4 g3) (bit 5 p3) (bit 6 p3))
              (and bc3 (bit 4 p3) (bit 5 p3) (bit 6 p3))))
        (setq c7 bc7)
        (setq c8 (or (bit 8 g3) (and bc7 (bit 8 p3))))
        (setq c9 (or (bit 9 g3) (and (bit 8 g3) (bit 9 p3))
              (and bc7 (bit 8 p3) (bit 9 p3))))
        (setq c10 (or (bit 10 g3) (and (bit 9 g3) (bit 10 p3))
              (and (bit 8 g3) (bit 9 p3) (bit 10 p3))
              (and bc7 (bit 8 p3) (bit 9 p3) (bit 10 p3))))
        (setq c11 bc11)
        (setq c12 (or (bit 12 g3) (and bc11 (bit 12 p3))))
        (setq c13 (or (bit 13 g3) (and (bit 12 g3) (bit 13 p3))
              (and bc11 (bit 12 p3) (bit 13 p3))))
        (setq c14 (or (bit 14 g3) (and (bit 13 g3) (bit 14 p3))
              (and (bit 12 g3) (bit 13 p3) (bit 14 p3))
              (and bc11 (bit 12 p3) (bit 13 p3) (bit 14 p3))))
        (setq c15 bc15)
        (setq p4 p3)
        (setq carry4 carry3))
```

Figure 3.10. MacPitts .mac Program for a 16-Bit 5-Stage Pipeline Adder Circuit (cont.)

```
;
;Stage Five
;
  (par (setq add0 (xor (bit 0 p4) carry4))
       (setq add1 (xor (bit 1 p4) c0))
       (setq add2 (xor (bit 2 p4) c1))
       (setq add3 (xor (bit 3 p4) c2))
       (setq add4 (xor (bit 4 p4) c3))
       (setq add5 (xor (bit 5 p4) c4))
       (setq add6 (xor (bit 6 p4) c5))
       (setq add7 (xor (bit 7 p4) c6))
       (setq add8 (xor (bit 8 p4) c7))
       (setq add9 (xor (bit 9 p4) c8))
       (setq add10 (xor (bit 10 p4) c9))
       (setq add11 (xor (bit 11 p4) c10))
       (setq add12 (xor (bit 12 p4) c11))
       (setq add13 (xor (bit 13 p4) c12))
       (setq add14 (xor (bit 14 p4) c13))
       (setq add15 (xor (bit 15 p4) c14))
       (setq carryout c15)
       (setq sum0 add0)
       (setq sum1 add1)
       (setq sum2 add2)
       (setq sum3 add3)
       (setq sum4 add4)
       (setq sum5 add5)
       (setq sum6 add6)
       (setq sum7 add7)
       (setq sum8 add8)
       (setq sum9 add9)
       (setq sum10 add10)
       (setq sum11 add11)
       (setq sum12 add12)
       (setq sum13 add13)
       (setq sum14 add14)
       (setq sum15 add15)
       (setq cout carryout))))
```

Figure 3.10. MacPitts .mac Program for a 18-Bit 5-Stage Pipeline Adder Circuit (cont.)

Initially, simulations on the 4-bit and 8-bit pipeline adder circuits could not be performed due to numerous wiring and alignment errors in the MacPitts designs. These errors are discussed in Chapter V of this thesis. After all of the wiring and alignment errors were corrected the two adders produced correct simulations using esim.

IV. DESIGN PERFORMANCE COMPARISONS

A. TIMING ANALYSIS USING CRYSTAL

1. Introduction

Crystal is a VLSI circuit delay analysis program developed at the University of California at Berkeley. The slowest paths in the circuit are determined by Crystal and this information can be used to calculate the maximum clock speed of the circuit. Version 2 of Crystal found in the berk85 VLSI design tools available on the UNIX VAX computer system was used for all timing and delay analysis.

Crystal reads circuit description information from a .sim file created by the circuit extractor program Mextra and then accepts commands from the programmer from the terminal keyboard. There are seven categories of Crystal commands and they must appear in the following order when a timing analysis is performed: model commands, circuit commands, dynamic node commands, check commands, setup commands, delay commands and miscellaneous commands. References 6 and 11 should be consulted for a complete listing of all Crystal commands and their use. Output from Crystal is written on the terminal screen and can be stored in a file if the UNIX "script" command is executed before the timing analysis is started.

2. Combinational Circuits

a. Performing a Delay Analysis

Combinational circuits are the easiest circuits to analyze using Crystal. First, all input and output pads should be labeled using the VLSI circuit editor Caesar. The label can be any combination of distinctive ASCII characters except space, tab, newline, double quote, comma, semi-colon and parenthesis and must not start or end with a number. Next a .sim file is created using Mextra with a -o option. Only four commands: "inputs", "outputs", "delay" and "critical" are necessary to analyze the circuit. The commands "inputs" and "outputs" are used to identify the input and output signals of the combinational circuit. Delay commands are used to tell Crystal when input signals change value [Ref. 6]. The form of the delay command is:

```
delay (signal name) tr tf
```

where tr is the time that the signal will rise to 1 and tf is the time that the signal will fall to 0. An example of a delay command is:

```
delay ain 3 0
```

This delay command specifies that the time that ain will rise is 3ns and the time that ain will fall is 0ns. This means that ain is initially set to 0 and will rise to 1 3ns later. If a negative time is used in a delay statement a transition of that signal will not occur after time 0. This allows the programmer to have input signals stable at the start of the timing analysis. The command "critical" directs Crystal to calculate the slowest path in the circuit.

Two other commands, "check" and "clear", may also be useful. The check command performs a static electrical check on the circuit. Information about nodes with no transistors connected to them, nodes that are not driven, nodes that don't drive anything, transistors that are permanently forced off, transistors connecting Vdd and GND, and transistors that are bidirectional is printed to the screen [Ref. 11]. All of this information, except for the information on the bidirectional transistors, is not very useful in a Macpitts generated circuit. This is because when the MacPitts silicon compiler does not use part of an organelle in a chip design the unused circuitry is left in the design resulting in improperly connected nodes and transistors. A bidirectional transistor is a transistor for which Crystal cannot determine the direction of signal flow within the transistor. To prevent Crystal from calculating circuit delays along impossible paths, bidirectional transistors must be labeled to show signal directions. (See paragraph 3.a. below for directions on how to label bidirectional transistors.)

The command "clear" is used to clear all previous delay information and critical calculations from Crystal.

Information on inputs and outputs is not affected. When a clear command is used new timing calculations can be made based on new delay commands for the same circuit.

Figure 4.1 shows the sequence of commands used to perform a timing analysis on a 1-bit combinational adder circuit. A check for bidirectional transistors was previously performed and none were found in this circuit. Line 2 shows the command used to invoke Crystal and lines 6 and 8 identify the circuit inputs and outputs. The Crystal output lines that are enclosed in brackets on lines 5 and 7 indicate that Crystal has completed execution of the previous commands. Crystal outputs a line in brackets after the execution of every command. In lines 10, 16 and 19 the two input bits, ain and bin, and the input carry bit, cin, are set to 1, 0 and 1, respectively, with delay commands. In lines 14, 17 and 20 Crystal indicates the number of stages that had to be examined to determine the timeing delay for each signal. After the delay commands, the critical command is given in line 22. Lines 23 through 55 shows the time delay through the critical path in the circuit. Each node that is in the critical path is identified with the time that it is driven. In this case the critical path started at input pad bin, goes through the combinational logic in the data-path and then ends at the output pad res 198.12ns later (see Figure 4.2).

```
% script
% crystal addc1.sim
Crystal, v 2
: build addc1.sim
[0:00.7u 0:00.2s 30k]
: inputs ain bin cin
[0:00.0u 0:00.0s 39k]
: outputs res
[0:00 0u 0:00.0s 39k]
: delay ain 0 -1
Marking transistor flow...
Setting Vdd to 1...
Setting GND to 0...
(28 stages examined.)
[0:00.2u 0:00.1s 48k]
: delay bin -1 0
(41 stages examined.)
[0:00.1u 0:00.1s 54k]
: delay cin 0 -1
(26 stages examined.)
[0:00.0u 0:00.0s 54k]
: critical
Node res is driven high at 198.12ns
    ...through fet at (885, 525) to Vdd after
  342 is driven high at 189.31ns
    ...through fet at (870, 457) to Vdd after
  357 is driven low at 179.77ns
    ...through fet at (849, 505) to GND after
  139 is driven high at 171.36ns
    ...through fet at (730, 387) to Vdd after
  258 is driven low at 85.70ns
    ...through fet at (668, 381) to 233
    ...through fet at (668, 376) to GND after
  221 is driven high at 81.04ns
    ...through fet at (623, 385) to Vdd after
  240 is driven low at 72.31ns
    ...through fet at (561, 379) to 225
    ...through fet at (561, 374) to GND after
  171 is driven high at 66.64ns
    ...through fet at (454, 387) to Vdd after
  255 is driven low at 48.79ns
    ...through fet at (392, 381) to 231
    ...through fet at (392, 376) to GND after
  219 is driven high at 44.13ns
    ...through fet at (347, 385) to Vdd after
  237 is driven low at 35.35ns
```

Figure 4.1. Crystal Delay Analysis of a 1-Bit Combinational Adder

```
47       ...through fet at (285, 379) to 223
48       ...through fet at (281, 374) to GND after
49     141 is driven high at 30.53ns
50       ...through fet at (264, 381) to Vdd after
51     119 is driven low at 13.17ns
52       ...through fet at (474, 603) to GND after
53     401 is driven high at 8.66ns
54       ...through fet at (518, 593) to Vdd after
55     bin is driven low at 0.00ns
56 [0:00.1u 0:00.2s 54k]
57 : clear
58 [0:00.0u 0:00.0s 54k]
59 : delay ain -1 0
60 Marking transistor flow...
61 Setting Vdd to 1...
62 Setting GND to 0...
63 (26 stages examined.)
64 [0:00.1u 0:00.1s 60k]
65 : delay bin 0 -1
66 (52 stages examined.)
67 [0:00.1u 0:00.1s 63k]
68 : delay cin -1 0
69 (61 stages examined.)
70 [0:00.2u 0:00.0s 63k]
71 : critical
72 Node res is driven high at 226.63ns
73       ...through fet at (885, 525) to Vdd after
74     342 is driven high at 217.82ns
75       ...through fet at (870, 457) to Vdd after
76     357 is driven low at 208.28ns
77       ...through fet at (849, 505) to GND after
78     139 is driven high at 199.87ns
79       ...through fet at (730, 387) to Vdd after
80     258 is driven low at 114.21ns
81       ...through fet at (668, 381) to 233
82       ...through fet at (668, 376) to GND after
83     221 is driven high at 109.55ns
84       ...through fet at (623, 385) to Vdd after
85     240 is driven low at 100.82ns
86       ...through fet at (561, 379) to 225
87       ...through fet at (561, 374) to GND after
88     171 is driven high at 95.15ns
89       ...through fet at (454, 387) to Vdd after
90     255 is driven low at 77.30ns
```

Figure 4.1. Crystal Delay Analysis of a 1-Bit Combinational Adder (cont.)

```
91      ...through fet at (392, 381) to 231
92      ...through fet at (392, 376) to GND after
93     219 is driven high at 72.64ns
94      ...through fet at (347, 385) to Vdd after
95     237 is driven low at 63.86ns
96      ...through fet at (285, 379) to 223
97      ...through fet at (281, 374) to GND after
98     141 is driven high at 59.04ns
99      ...through fet at (264, 381) to Vdd after
100    170 is driven low at 42.03ns
101     ...through fet at (188, 372) to GND after
102    63 is driven high at 28.83ns
103     ...through fet at (182, 189) to Vdd after
104    36 is driven low at 15.52ns
105     ...through fet at (885, 364) to GND after
106    148 is driven high at 8.65ns
107     ...through fet at (875, 408) to Vdd after
108    cin is driven low at 0.00ns
109 : quit
```

Figure 4.1. Crystal Delay Analysis of a 1-Bit Combinational Adder (cont.)

When Crystal does a timing analysis of a clocked circuit it is assumed that each clock phase (or clock period segment in the case of a MacPitts design) is long enough for the combinational logic in the circuit to settle. But in a MacPitts circuit the first and second clock period segments, t1 and t2, are used for the settling time of the combinational logic. Crystal will give an overly long delay for t1 of a MacPitts design because all of the logic propagation delay will be assigned to this section.

Another problem is that it will not be possible to determine the logic delay of any stage in the pipeline if the delay of the clock phase signals phia, phib and phic getting to the registers or flags is longer than the stage logic delays. This is because Crystal only gives the timing delay for the critical or longest path in the circuit.

The problems are solved by dividing the timing analysis of the Macpitts pipeline design into two parts. First the clocked registers and flags of the chip are analyzed for the timing delay of the input clock phase signals and then the combinational logic in each pipeline stage is analyzed to determine the slowest stage in the pipeline system.

(2) Register and Flag Delays. The first step in performing a timing analysis of the clocked registers and flags is to edit the MacPitts circuit using Caesar. All

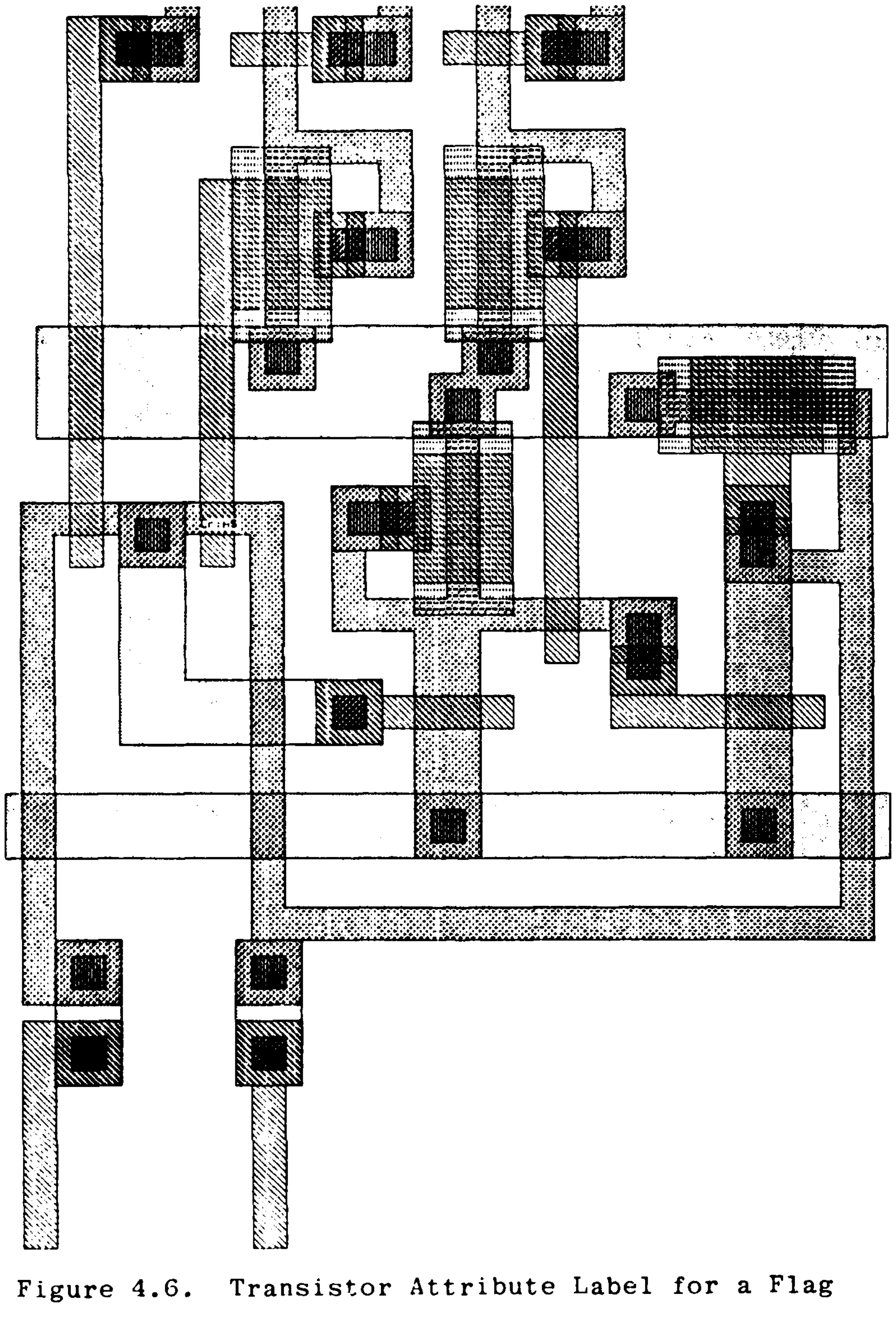


Figure 4.6. Transistor Attribute Label for a Flag

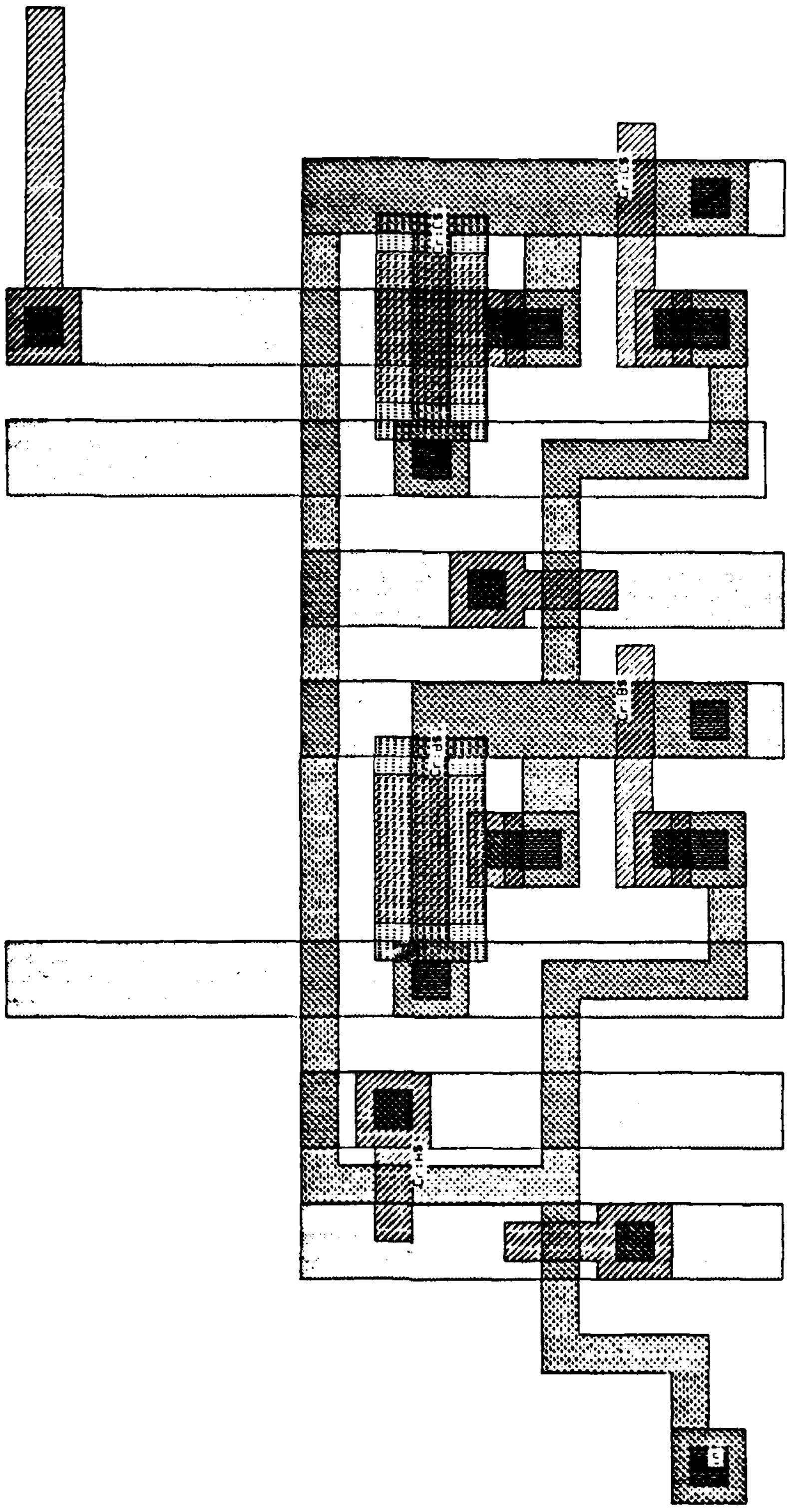


Figure 4.5. Transistor Attribute Labels for a Register Cell

placed on each transistor on the side of the gate that shares the electrical connection to the other transistors. Bidirectional transistors that are not electrically connected should have different labels.

Figure 4.5 shows the stipple plot of a register cell that has five bidirectional transistors labeled with transistor attributes. The bidirectional transistor labeled Cr:A$ is not electrically connected to any other bidirectional transistor. The source side of the gate has been labeled. The two transistors labeled Cr:B$ are the pull-up and pull-down transistors of an inverter. Due to the unusual MacPitts inverter structure Crystal could not determine the direction of signal flow and identified the pull-up and pull-down transistors as bidirectional. Since both of the transistors are electrically connected the same transistor attribute label has been placed on the side of the gates that are connected. Transistors labeled Cr:C$ are the pull-up and pull-down transistors of another inverter. Figure 4.6 shows the transistor attribute labeling for the one bidirectional transistor in a flag.

b. Crystal Commands for Clocked Circuits

(1) Problems Analyzing a MacPitts Design. Crystal was designed to be used for a non-overlapping clocking scheme. The overlapping clock phases and the five segment period of the MacPitts clock (see Figure 2.2) make the MacPitts pipeline adder circuit much more difficult to analyze.

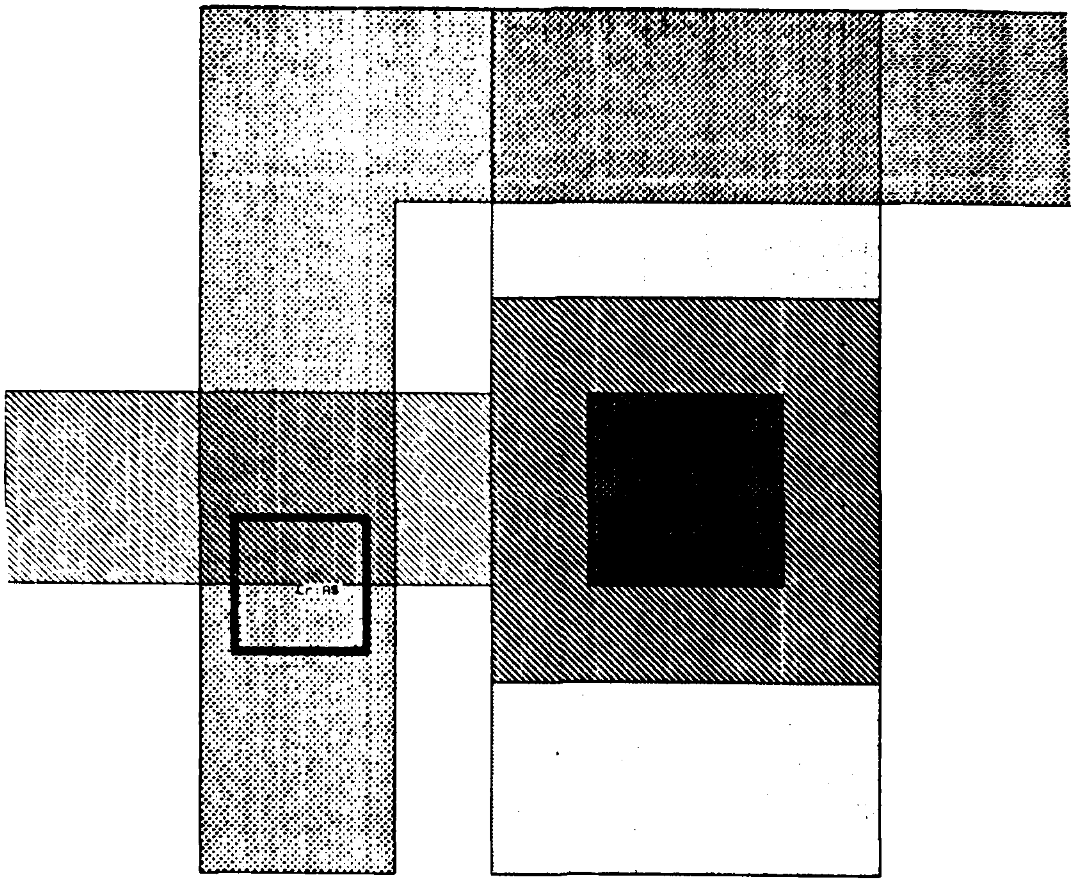


Figure 4.4. Placement of a Transistor Attribute Label on a Bidirectional Transistor

Macpitts data storage elements, the register and the flag, each have bidirectional transistors in them. The register has five bidirectional transistors in each register cell and the flag has one.

The procedure used to show the direction of signal flow through a bidirectional transistor is to attach a transistor attribute label to the transistor using Caesar. A transistor attribute label has the following form:

```
Cr:(label)$
```

The label must be placed exactly in the middle of the source or drain edge of the gate region of the transistor. This is done by placing the center of the Caesar bounding box over the center of the source or drain edge of the gate and typing the following Caesar command:

```
: la Cr:(label)$ center
```

Figure 4.4 shows a stipple plot of a bidirectional transistor. The center of the bounding box is on the center of the source edge of the gate region and the transistor attribute label Cr:A$ has been affixed to this point.

If a bidirectional transisotr is not electrically connected to any other bidirectional transistor the transistor attribute label should be placed on the source edge of the gate. If two or more bidirectional transistors are electrically connected the same attribute label should be

TABLE II

COMPARISON OF MEAD-CONWAY AND CRYSTAL DELAY CALCULATIONS

	Mead-Conway	Crystal
logic delay	56ns	93.79ns
wire delay	95ns	105.84ns
pad delay	21ns	27ns
total delay	172ns	226.63ns

up by 2.25. Multiplying the above wire delay by 2.25 gives a total wire delay of 95ns.

(4) Pad Delays. The signal delay for the output pad is approximately 13ns [Ref. 13]. Due to the lack of available information on the signal delay for the input pad the delay calculated by Crystal of 8ns will be used in this comparison. This gives a total pad delay of 21ns.

(5) Comparison of Results. In Table II a comparison is given of the circuit delays calculated using the Mead-Conway methods and those calculated by Crystal. The logic delays calculated using the Mead-Conway methods are less than that calculated by Crystal because delays caused by the polysilicon wires connecting the gates together in the data-path are not taken into account in the Mead-Conway calculations. The total circuit delay of 172ns calculated by the Mead-Conway methods is in close agreement with the 226.63ns delay calculated by Crystal. It can be concluded that the circuit delay information given by Crystal is accurate and can be used with confidence.

3. Pipeline Circuits

a. Labeling Bidirectional Transistors

Before a timing analysis can be done on a MacPitts design the bidirectional transistors in the circuit must be identified by using the check command of Crystal and properly labeled so that Crystal does not have to determine the direction of signal flow through these transistors. The time, is 142t. Reference 12 states that the signal transit time equals 0.3ns for a six micron design (lambda equals 3 microns) and the 1-bit combinational adder is a 4 micron design (lambda equals 2 micron). The transit time is scaled down by dividing by the scale factor 1.5 (6 microns divided by 4 microns). This gives a transit time of 0.2ns. Using this value, a logic delay of 28ns is obtained. This value is doubled to account for stray capacitance in the circuit giving a total logic delay of 56ns.

(3) Wire Delays. From Figure 4.2 it can be seen that there are long metal and polysilicon runs in the circuit. The total length of metal runs from the input pad to the Weinberger array and from the data-path to the output pad is approximately 3.9mm. The total length of polysilicon runs from the input pad to the Weinberger array, from the Weinberger array to the data-path and from the data-path to the output pad is approximately 2.1mm. There are no significant diffusion runs in the circuit.

Reference 12, page 231, states that metal line delays equal 0.1ns/10mm and that polysilicon line delays equal 200.0ns/10mm. Using these values a wire delay of 42ns is calculated. The wire delays used in the above calculations are based on a 6 micron design. When lambda is scaled down the capacitance per unit length of wire stays constant but the resistance scales up quadratically. Since lambda is scaled down by a factor of 1.5 the wire resistance scales

TABLE I

RESULTS OF LOGIC DELAY CALCULATIONS

LOGIC ELEMENT	k	f	GATE DELAY	# OF GATES	TOTAL DELAY
inverter	4	1	4t	1	4t
pass transistor	-	-	2t	1	2t
nand gate	8	3	48t	1	48t
nand gate	4	2	16t	3	48t
nand gate	4	1	8t	5	40t

TOTAL LOGIC DELAY = 142t

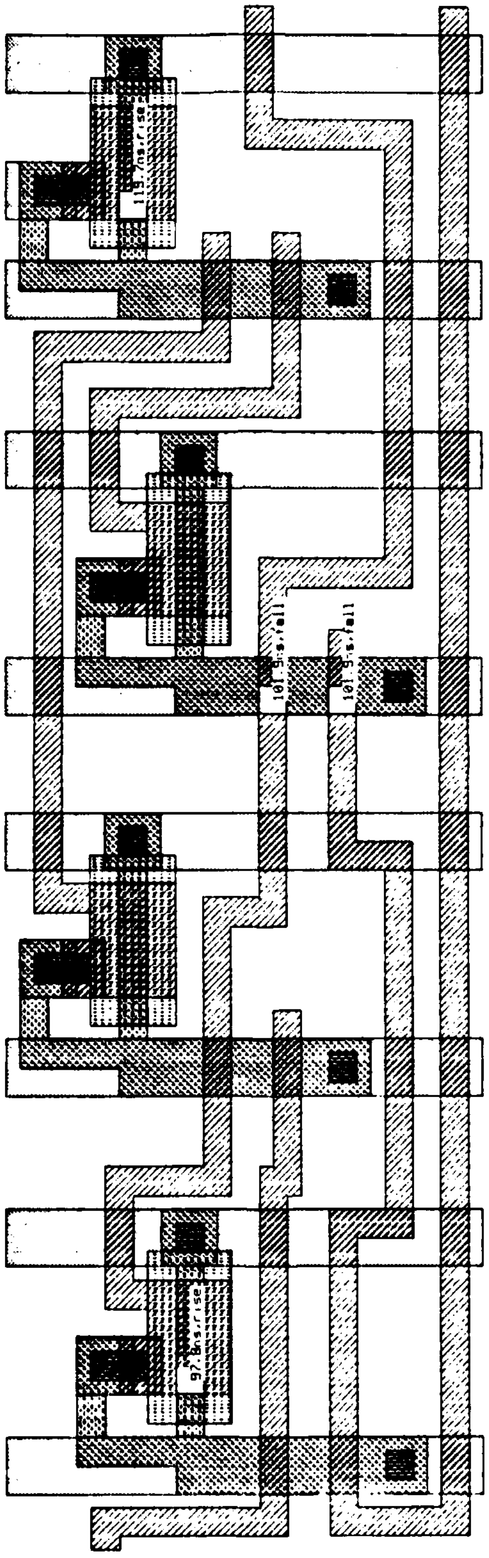


Figure 4.3. Caesar Display of Crystal Generated Timing Delay Information

critical path is identified on the Caesar display screen along with the timing delay information. Figure 4.3 shows an example of how the timing delay information is displayed.

b. Validation of Crystal's Timing Data

(1) Introduction. Previous to this research effort there had been no experience at the Naval Postgraduate School in using Crystal to analyze circuits. The accuracy of the results produced by Crystal was not known. In order to gain confidence in Crystal a complete timing analysis of the 1-bit combinational adder previously analyzed by Crystal was performed using the Mead-Conway guidelines in [Ref. 12].

The critical path found by Crystal was used to determine which transistors in the circuit were on. The delay calculations are divided into logic delays, wire delays and pad delays.

(2) Logic Delays. The following equations were used to calculate the logic delay in the circuit:

$$\begin{aligned} Tpt &= 2t \\ Tinv &= fkt \\ Tnand &= 2fkt \end{aligned}$$

where Tpt is the delay for a pass transistor, Tinv is the delay for an inverter, Tnand is the delay for a nand gate, t is the signal transit time, f is the gate fanout, and k is the pull-up to pull-down transistor ratio. Table I shows that the total logic delay, in terms of the signal transit

In line 57 the clear command is used so that a new timing delay analysis on the same circuit can be made. In lines 59, 65 and 68 new delay commands set ain, bin and cin to 0, 1 and 0 respectively. The critical command is given on line 71 and new critical path information is shown on lines 72 through 108. This time the critical path starts at the input pad cin, goes through the Weinberger array and the combinational logic in the data-path and ends at the output pad res 226.63ns later. After finishing a Crystal timing analysis the command "quit" should be used to exit the Crystal program.

As can be seen from the timing analysis of the 1-bit combinational adder, the longest critical path occurs when cin is driven to a low state. This is because the cin signal must travel through the Weinberger array and the first organelle in the data-path. This circuitry is normally at a high state unless brought low by a low cin. A high cin causes no level transitions so there is no delay through the circuitry. For a low cin there is a low transition that takes approximately 30ns to propagate through the Weinberger array and the first organelle in the data-path.

If the -g (filename) option is used with the critical command [Ref. 11] the critical path timing information is printed in (filename) in a format that can be accessed by Caesar using the Caesar "source" command. Each node in the

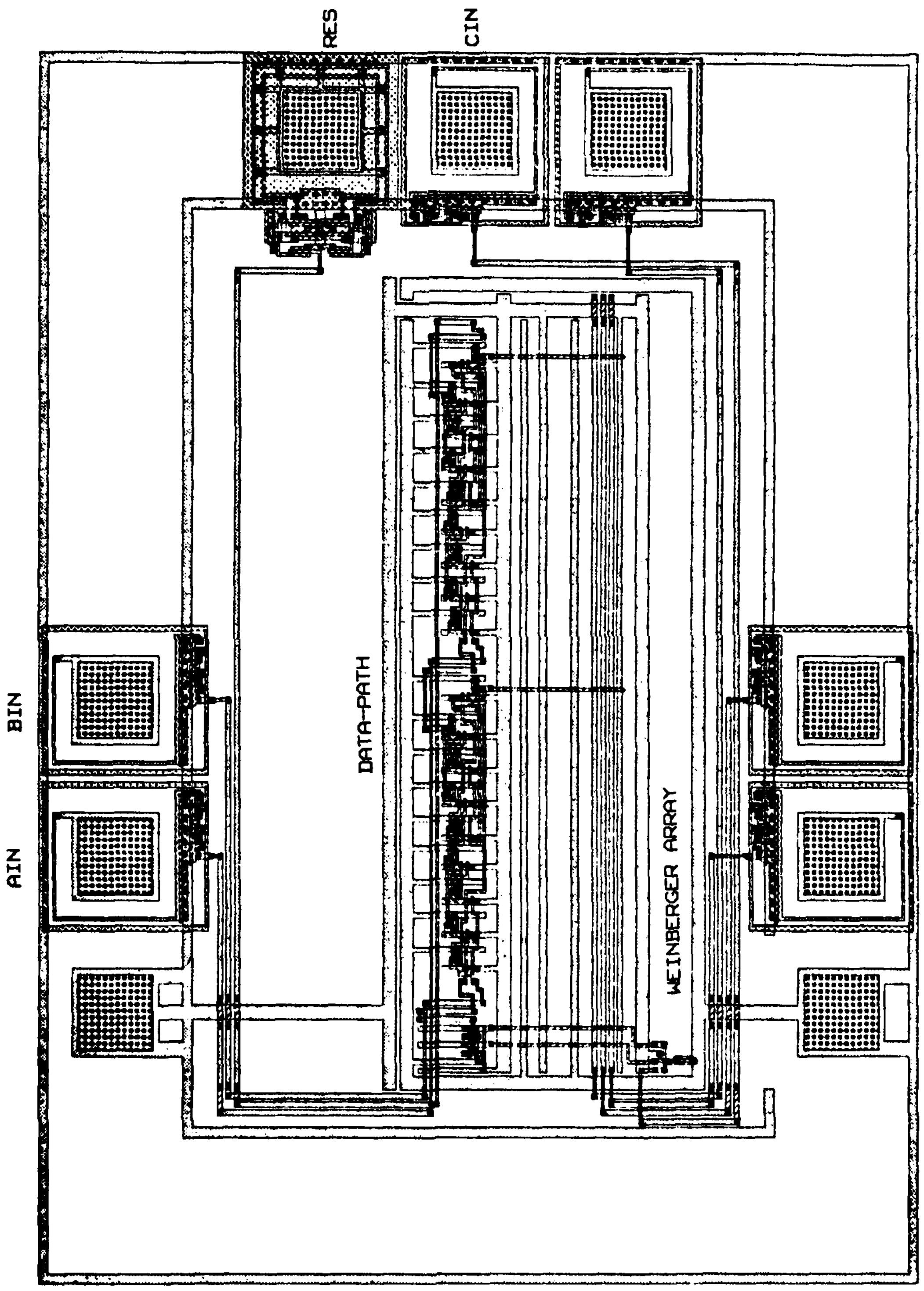


Figure 4.2. 1-Bit Combinational Adder

logic except the flags block, registers and the ground, power and clock pads is deleted from the circuit. This is done so that Crystal does not use the extraneous circuitry in determining the critical path through the registers and flags. Next, the registers are deleted from the circuit because the clock phase signals will take longer to reach the flags than the registers. This is because the registers are closer to the clock pads on the clock bus and also the clock phase signals are further delayed in the flag block by two inverters. Finally, the input and output lines of each flag are disconnected from the extraneous data lines going to the Weinberger array, if not already done so, and the input and output wires of each flag are labeled (see Figure 4.7). Figure 4.8 shows what the edited circuit looks like for 4-bit 5-stage pipeline adder.

The timing analysis of a clocked circuit is similar to that of a combinational circuit except that there is a separate set of delay and critical commands for each clock phase. For the MacPitts overlapping clock there is a separate set of delay and critical commands for each of the five segments of the clock period. The clear command is used between each set of delay and critical commands. Prior to the delay commands, the clock phases that do not change state during a section of the clock period should be set to the high or low state using the set command [Ref. 11]. Inputs that are set to a state are not used by Crystal to determine

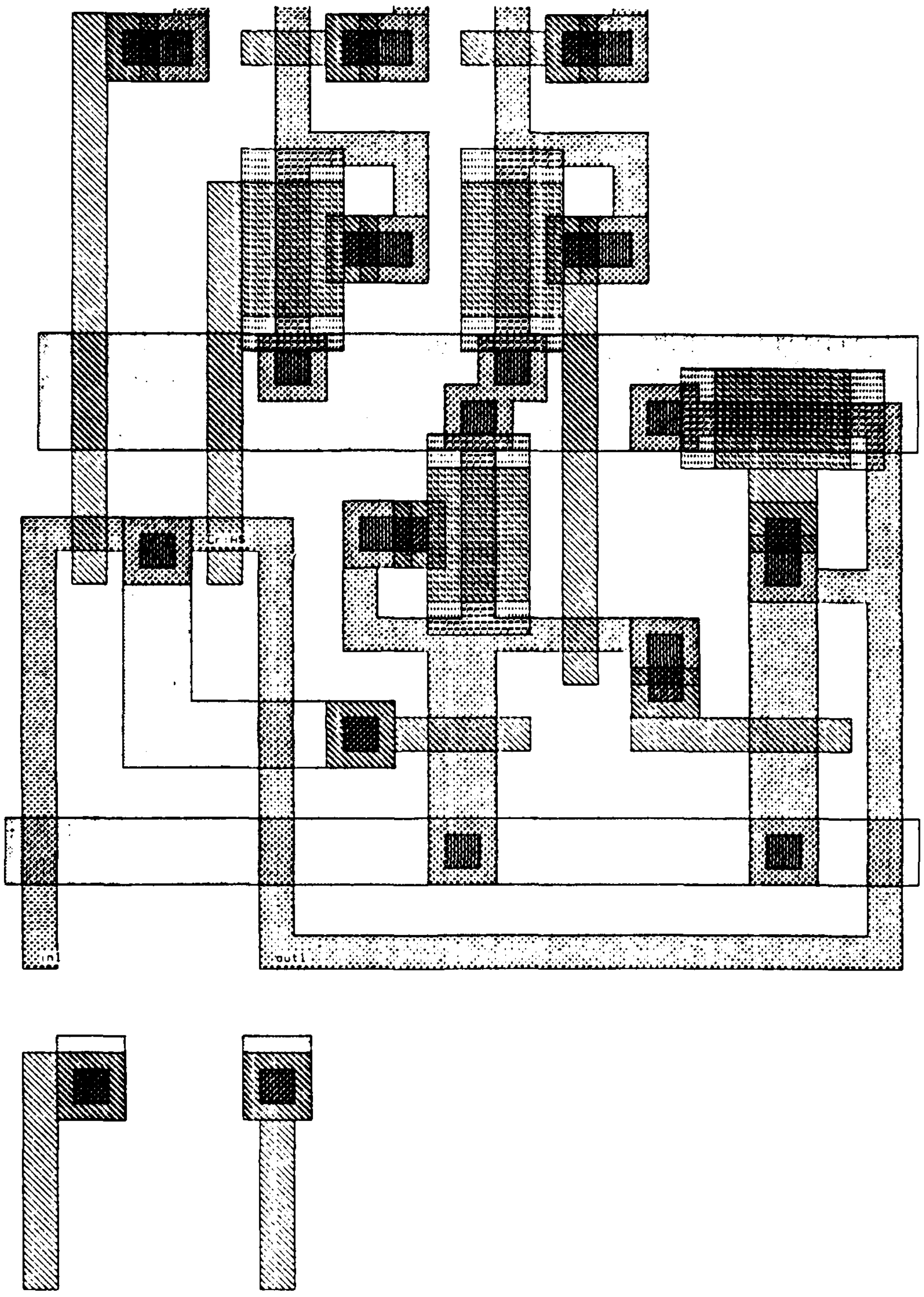


Figure 4.7. Disconnecting the Flag Input and Output Lines

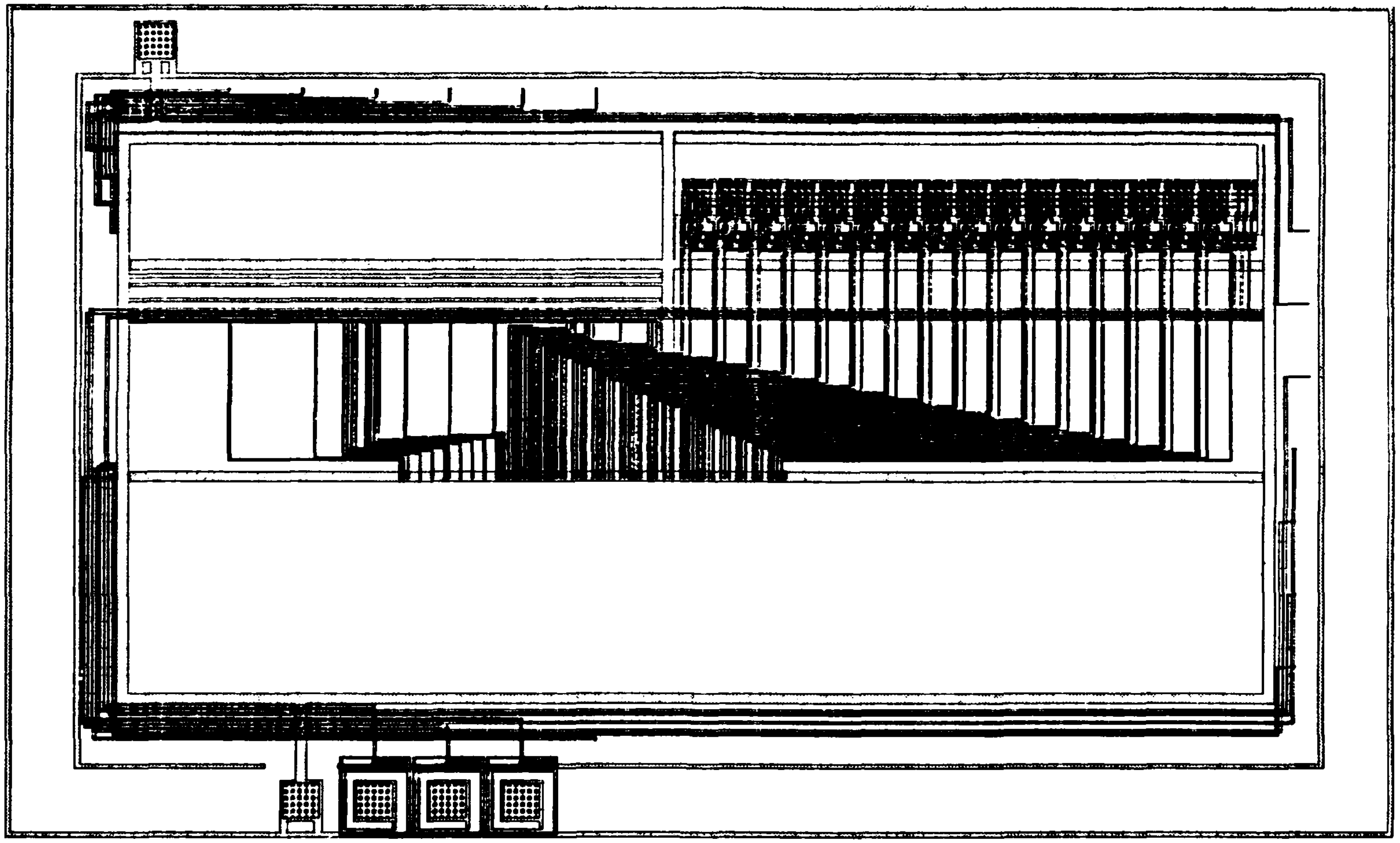


Figure 4.8. 4-Bit 5-Stage Pipeline Adder Edited for Crystal Analysis

the critical path because they do not have a state transition. Also, if delay commands for inputs other than the clock phases are not used Crystal assumes that the input signals stabilize long before the start of the clock period. Crystal then determines the longest critical path in the circuit no matter what the state of the non-delayed inputs are. In Figure 4.9 the Crystal commands used to analyze the clock phase delays through the flags block are listed.

(3) Pipeline Stage Delays. A separate Crystal timing analysis must be performed on the combinational logic in each pipeline stage in order to obtain propagation delays for each stage. First, the input and output signals of each stage must be determined. Input signals come from input pads or from register or flag outputs. Output signals are inputs to registers, flags or output pads. Next, using Caesar, the input and output lines of each stage are disconnected from any logic elements that are not part of that stage. This is done so that Crystal does not use circuitry that is not part of a stage in determining the critical path through that stage. Labels are then placed on all input and output lines.

Figures 4.10 and 4.11 show two different circuits before they are edited using the above procedure and Figures 4.12 and 4.13 show the circuits after they have been edited. In Figure 4.12 node cl is the output line of stage 1 of the pipeline and has been disconnected from the

```
% script
% crystal addp4.sim
: inputs in<16:1> phia phib phic
: outputs out<16:1>
: set 1 phia phic
: delay phib 0 -1
: critical
: clear
: set 1 phia
: delay phib -1 0
: delay phic -1 0
: critical
: clear
: set 0 phib phic
: delay phia -1 0
: critical
: clear
: set 0 phib phic
: delay phia 0 -1
: critical
: clear
: set 1 phia
: set 0 phib
: delay phic 0 -1
: critical
: quit
```

Figure 4.9. Crystal Commands: Timing Delay of Clock Phases

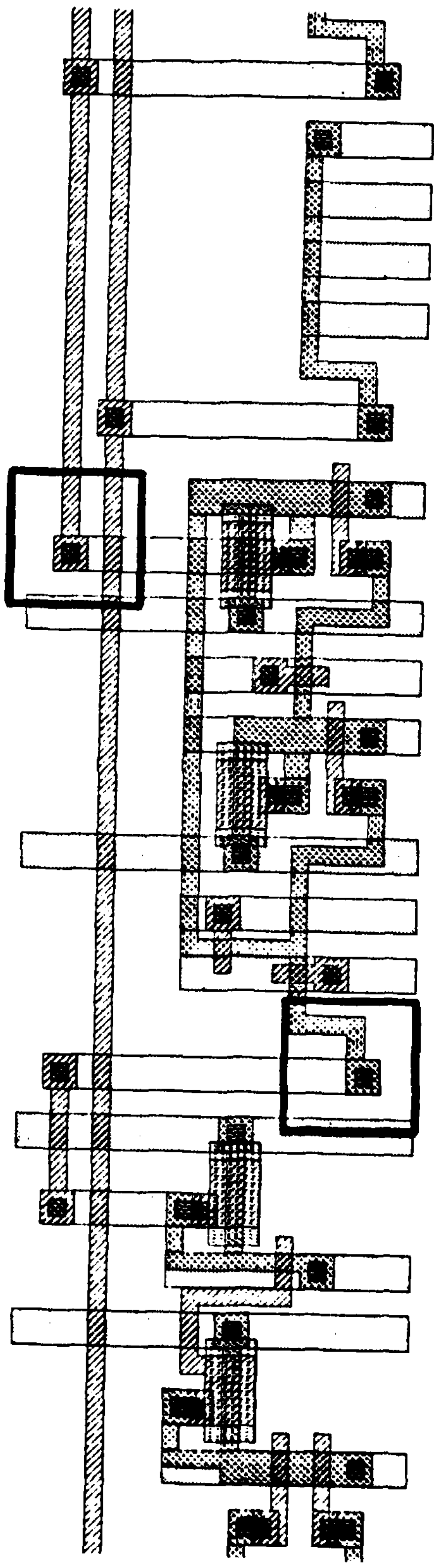


Figure 4.10. A Register Cell Before Editing

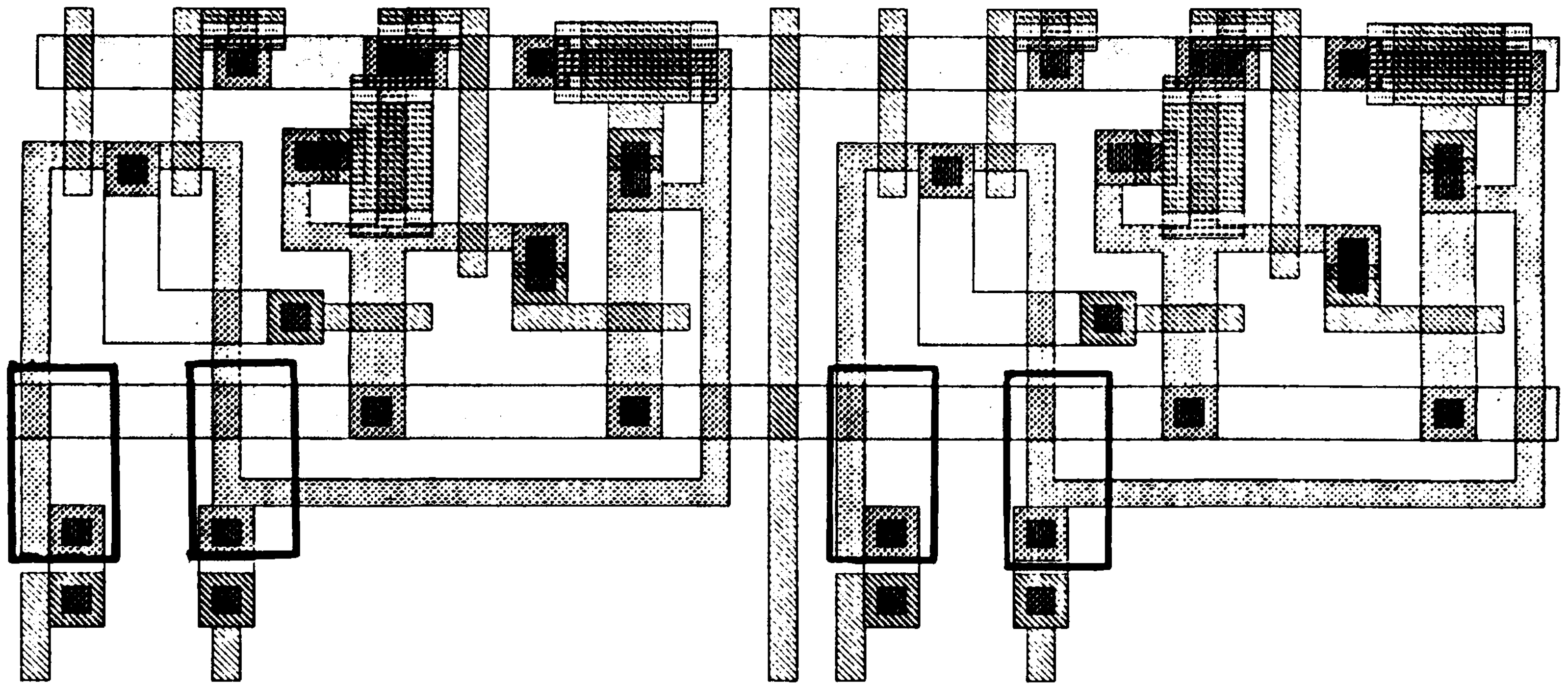


Figure 4.11. Flag Cells Before Editing

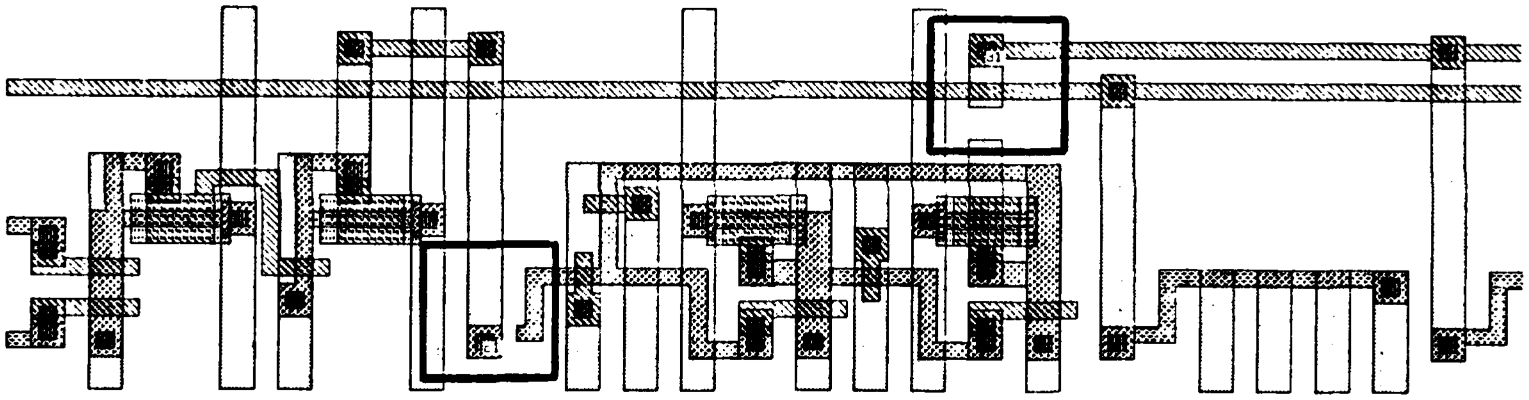


Figure 4.12. A Register Cell After Editing

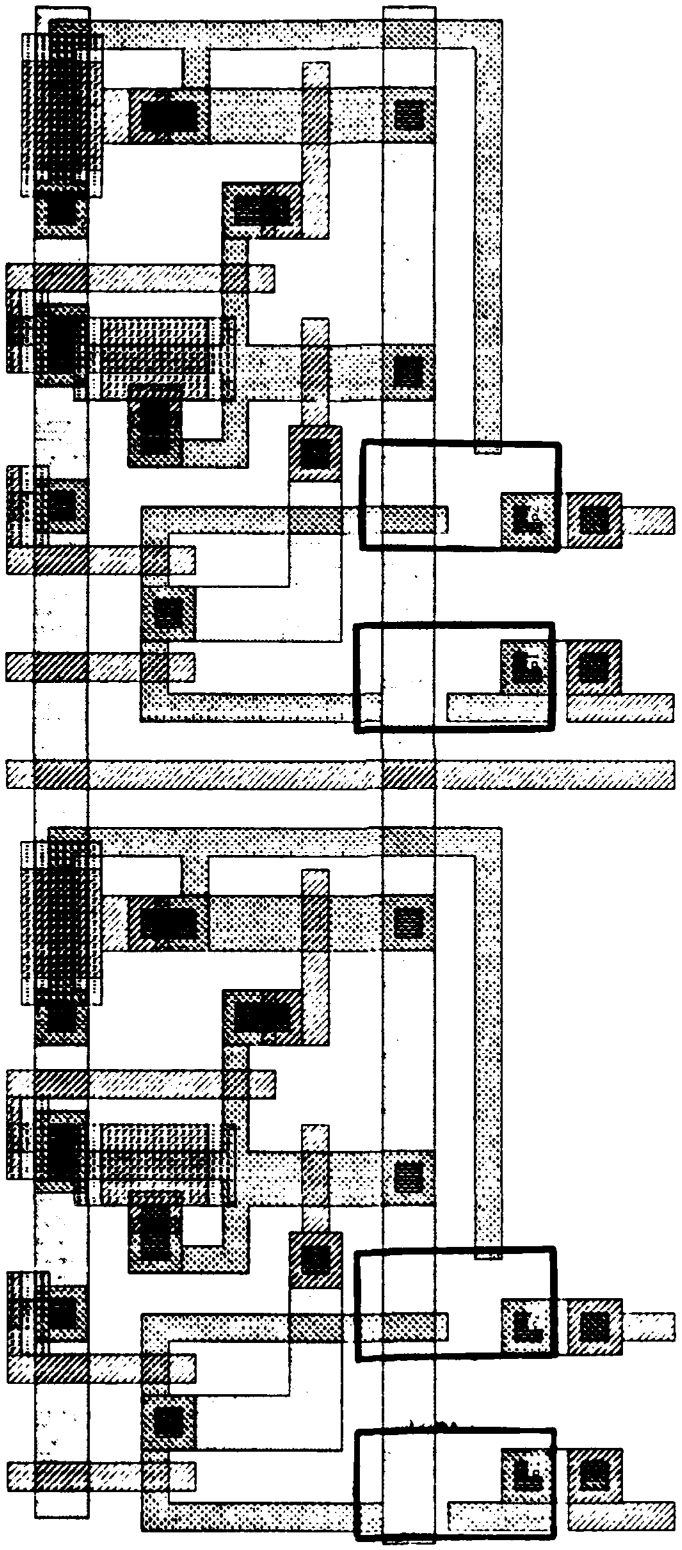


Figure 4.13. Flag Cells After Editing

input of the storage register cell. Node d1 is the input line of stage 2 and has been disconnected from the output of the register cell. In Figure 4.13 nodes o1 and p1 are output lines of stage 4 and have been disconnected from the input lines of the storage flags. Nodes o2 and p2 are inputs of stage 5 and have been disconnected from the output lines of the flags.

After all stages have been isolated and input and output lines labeled a .cif file is created using Caesar and then a .sim file is created using Mextra. A Crystal timing analysis is then performed on each stage in the pipeline using the same procedure as used when performing an analysis on a combinational logic circuit.

B. DESIGN COMPARISONS

Three important parameters used when comparing the performance of integrated circuit designs are chip size, power and speed.

In order to determine the speed of a MacPitts pipeline design the logic delay in each stage and the clock phase delays must be compared. The propagation time of the slowest stage in the pipeline is compared to the sum of the first two segments of the clock period t1 and t2. This is because all logic propagation in the circuit must be settled before t3 when the inputs to all storage registers and flags are sampled. The slowest of these times is then added to t3, t4 and t5 to determine the clock period.

Table III shows the propagation delay for each stage of a 4-bit pipeline adder and an 8-bit pipeline adder (4 micron designs). The long delays in stages 2 through 5 of each adder are caused by long delays through the Weinberger array and the long high-resistance polysilicon runs carrying data from the registers to the array and carrying data back and forth from the flags block to the array. The delays through the Weinberger array are due to three factors. First, the inputs to the array from the registers and flags are driven by k=4 inverters. These inverters, which are not super buffered, drive up to five nor gates in the array thus adding substantial delay to the stage [Ref. 12]. This delay could be considerably reduced if the outputs of all registers and flags were super buffered. Second, the propagation delay in the array is high due to the large number of nested NOR gates in the array. In some cases up to five NOR gates are nested to perform a particular function (i.e. an XOR function). This is much more delay than would be found in the two level nesting of a PLA. The excessive delays in the array are also caused by the long polysilicon lines that connect the inputs and outputs of the NOR gates. In some cases an output of a NOR gate is connected to the input of another NOR gate by a polysilicon wire that runs nearly the total width of the array. The increase in stage propagation delay of the 8-bit adder when compared to the 4-bit adder is due to the increased size of the Weinberger

TABLE III

PIPELINE STAGE DELAY

STAGE	4-BIT PIPELINE ADDER	8-BIT PIPELINE ADDER
1	33.59ns	51.87ns
2	126.14ns	255.53ns
3	106.60ns	222.89ns
4	142.70ns	250.63ns
5	141.63ns	203.87ns

array of the 8-bit adder and not due to a poorly designed pipeline chip.

In Table IV the delay in each clock period segment is listed. The long delays are due to the input clock pads not being super buffered. One k=4 inverter on each clock pad must drive eight k=4 inverters; one inverter for each of the seven registers and one input inverter to the flags block. Each of the input inverters of the registers and flag block cause further delay because they are not super buffered but must drive many register cells and flags. In the case of the 8-bit pipeline adder one k=4 inverter must drive twenty-seven flags. Additional delay is caused by the long clock bus. The clock signals must traverse a length nearly equal to the height and width of the chip before reaching the flags block. If the clock input pads, the input inverters, all registers and the flags block were super buffered the timing delay of each clock period segment would be substantially improved.

Comparing Tables III and IV it can be seen that the propagation delays through clock period segments t1 and t2 are greater than the slowest stage for both the 4-bit and 8-bit pipeline adders. Thus, the clock period is found by adding t1 through t5. The clock period of the 4-bit 5-stage pipeline adder is 486.74ns (2.055 MHz clock) and the clock period of the 8-bit 5-stage pipeline adder is 706.32ns (1.415 MHz clock).

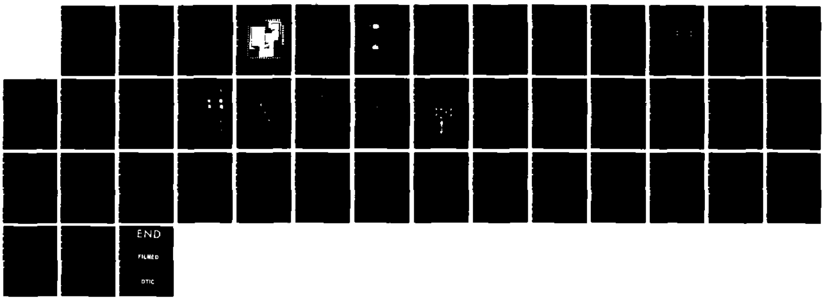
TABLE IV

CLOCK SIGNAL DELAY

CLOCK PERIOD SEGMENT	4-BIT PIPELINE ADDER	8-BIT PIPELINE ADDER
t1	116.00ns	170.46ns
t2	66.62ns	102.66ns
t3	82.93ns	106.96ns
t4	100.87ns	153.56ns
t5	120.05ns	172.68ns

AD-A158 995 SILICON COMPILER DESIGN OF COMBINATIONAL AND PIPELINE ADDER INTEGRATED CIRCUITS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA A O FROEDE JUN 85 2/2

UNCLASSIFIED F/G 9/5 NL



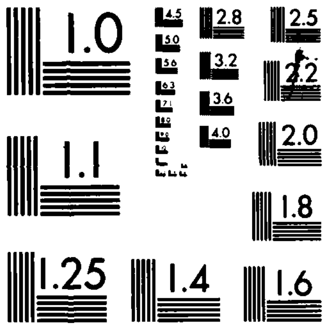

1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Table V lists the chip size, power and speed of several 4 micron combinational and pipeline adder circuits designed by MacPitts. In addition, a 16-bit 4-stage pipeline adder that was designed by hand is also listed [Ref. 5]. (See Figure 4.14.)

Chip size and worst case static power consumption are standard outputs from the MacPitts silicon compiler. The required power for the hand designed adder was found by using a program called powest that makes an estimate of the DC power required in a circuit based on the number of enhancement and depletion mode transistors in the circuit. Powest uses a .sim file as input and an output of the average DC power (based on one-half of the transistors being on at any time) and the maximum DC power (based on all transistors being on) is printed on the terminal screen. The value of power listed in Table V for the hand designed adder is the maximum DC power. The command to run powest is:

```
powest -p < filename.sim
```

For comparison, powest was run on all of the MacPitts designs and the power estimates calculated by powest and MacPitts were, on the average within 10% of each other.

All chip speed values listed in Table V were calculated by Crystal. Reference 5 estimates the clock speed of the 16-bit 4-stage pipeline adder as 8 MHz. This is seven times faster than the 1.141 MHz calculated by Crystal. The reason

TABLE V

PERFORMANCE COMPARISONS OF DESIGNS

Design	Chip Size mm	Chip Area mm2	Number Of Transistors	Power W	Speed MHz
1-bit combinational adder	2.026 x 1.462	2.962	122	0.035448	4.8
4-bit combinational adder	2.292 x 2.398	5.496	362	0.090291	3.38
8-bit combinational adder	3.508 x 3.614	12.678	682	0.163414	2.05
4-bit pipeline adder	4.828 x 2.918	14.088	1047	0.199269	2.055
8-bit pipeline adder	6.650 x 4.358	28.981	1786	0.356321	1.415
16-bit 4-stage pipeline adder [Ref 5]	6.250 x 6.750	42.188	1233	1.052144	1.141

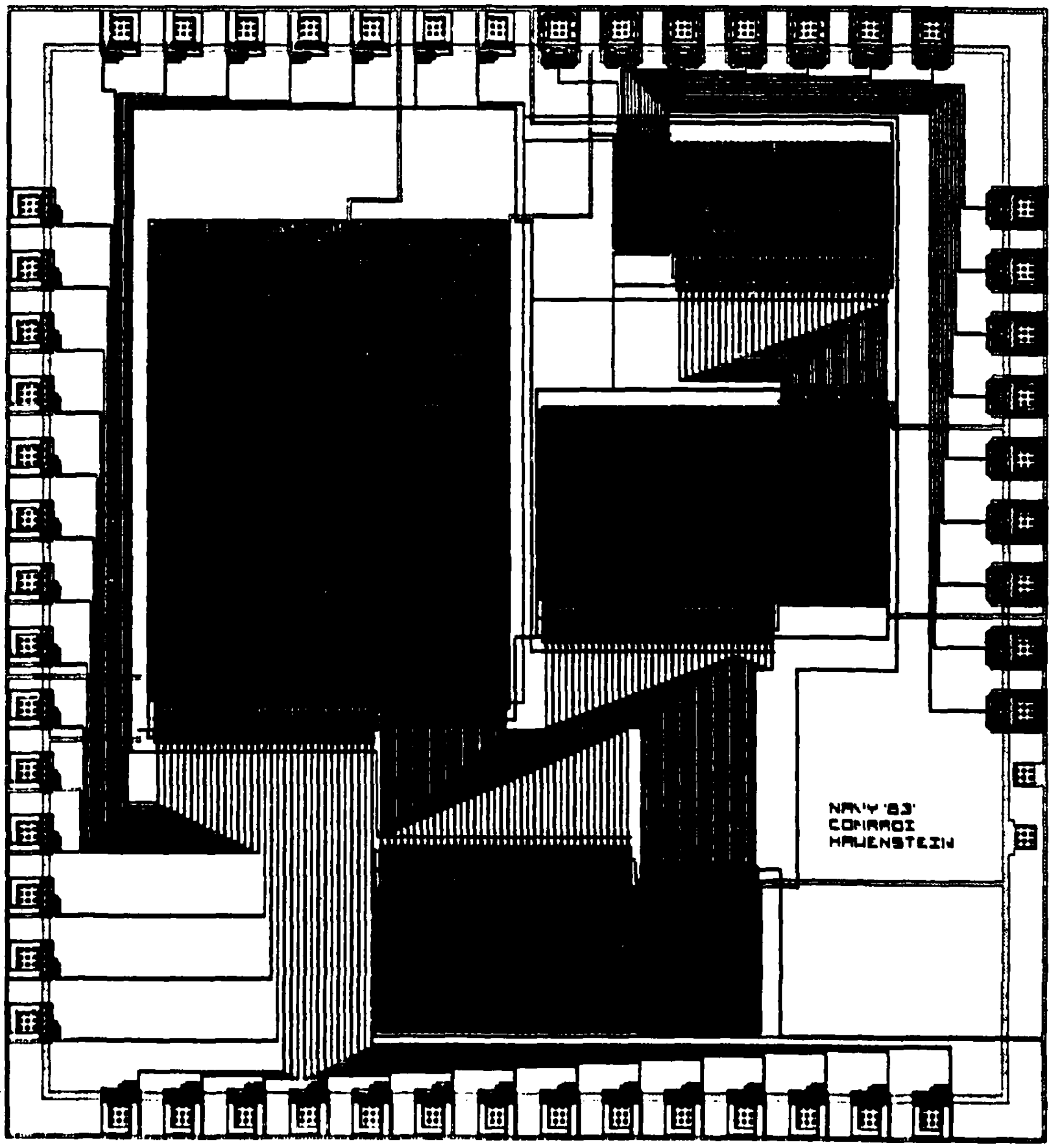


Figure 4.14. Hand Designed 16-Bit 4-Stage Pipeline Adder

for the discrepancy is that reference 5 does not take in account that the clock pads in the circuit, which are not super buffered, must drive a large number of pass transistors. Clock pad phia drives 138 pass transistors that clock data into the five PLAs in the circuit while clock pad phib drives 121 pass transistors that clock data out of the PLAs.

Another interesting observation about the hand designed circuit from reference 5 is that when the circuit is examined using Caesar a misalignment of one-half lambda between the data, power and ground buses going into the PLAs and the PLA blocks is found. As seen in Figure 4.15, the bus misalignments are not enough to disconnect any wires.

As expected, when the combinational adder circuits were converted to pipeline circuits the chip size and power increased, but the increase in chip throughput (or speed) anticipated in a pipeline design did not occur. The slow circuitry of the Weinberger array, non-super buffered clock pads and long polysilicon runs in the MacPitts pipeline circuits caused excessive delays and decreased performance below that of the combinational circuits. The excessive delays could be reduced if the Weinberger array was redesigned to reduce the NOR gate nesting or replaced by a PLA, if all input lines to the array were super buffered and if the long polysilison runs were replaced with metal or diffusion runs. If the design of a 16-bit pipeline adder were possible it is expected that this design would have a clock speed less than

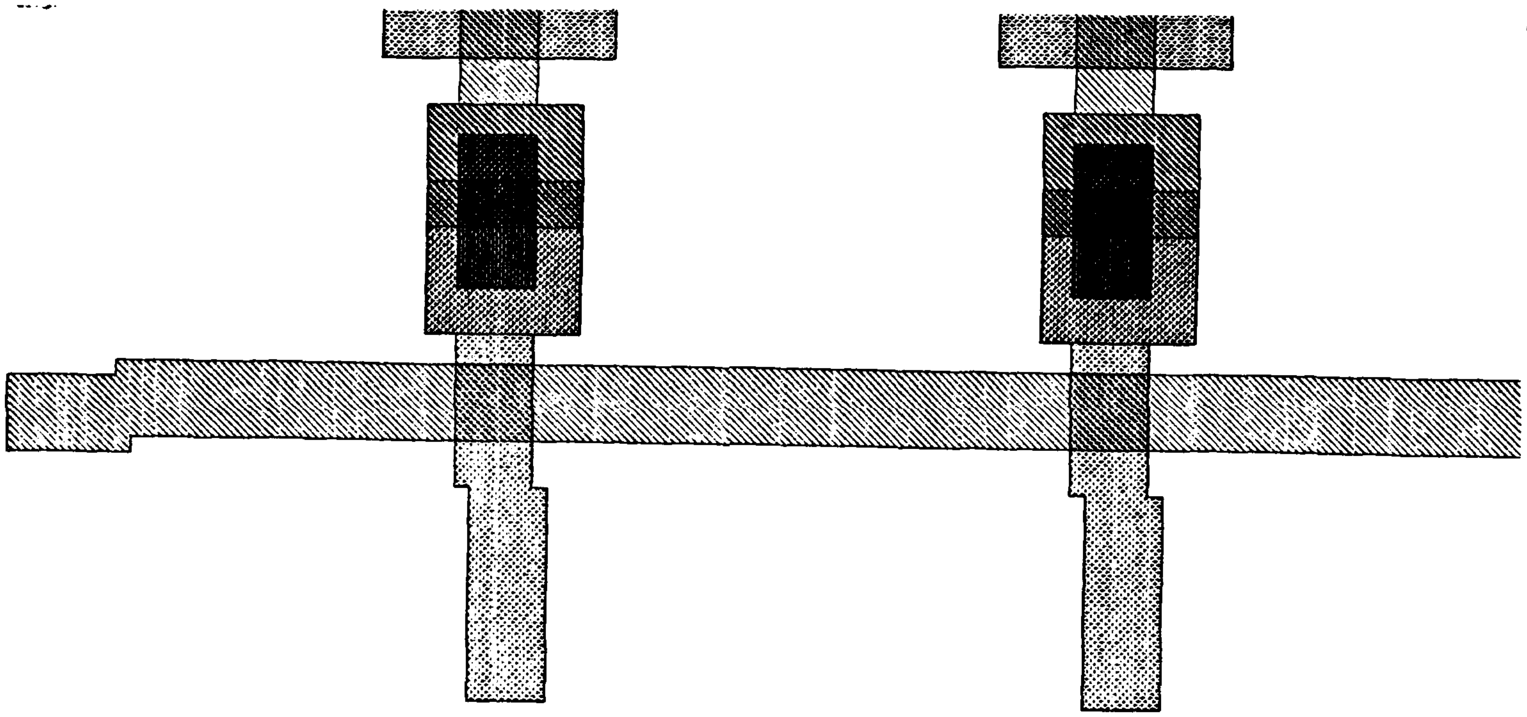


Figure 4.15. Misalignment of Chip Buses and PLA for 16-Bit 4-Stage Pipeline Adder Circuit

and chip area much larger than the hand designed adder. Even with fast logic in each pipeline stage and super buffered clocks the fact that the last three segments of the MacPitts clock period cannot be used for logic propagation insures that the MacPitts pipeline designs will be slower than any well designed hand-crafted circuit.

V. MACPITTS DESIGN ERRORS

A. INTRODUCTION

Although the MacPitts silicon compiler is expected to generate error free designs, several cases have been found where design errors have been made. These design errors fall into two categories: wiring errors and alignment errors. Wiring errors have occurred when wires become electrically connected when they should not be and alignment errors have occurred when circuitry has been placed incorrectly on the chip so that it does not align properly with adjacent circuitry.

B. WIRING ERRORS

1. Description of Errors

A case of a fatal wiring error was discovered where the MacPitts compiler electrically connected all three clock lines that run in the clock bus below the data-path to a data line that was running from the data-path to the Weinberger array. This error was found to occur whenever the last organelle of the data-path or sequencer is the organelle used by the compiler to transfer data from the data-path to the Weinberger array (see Figure 5.1). The vertical polysilicon data wire of this organelle runs parallel and only four lambda away from a large ground bus

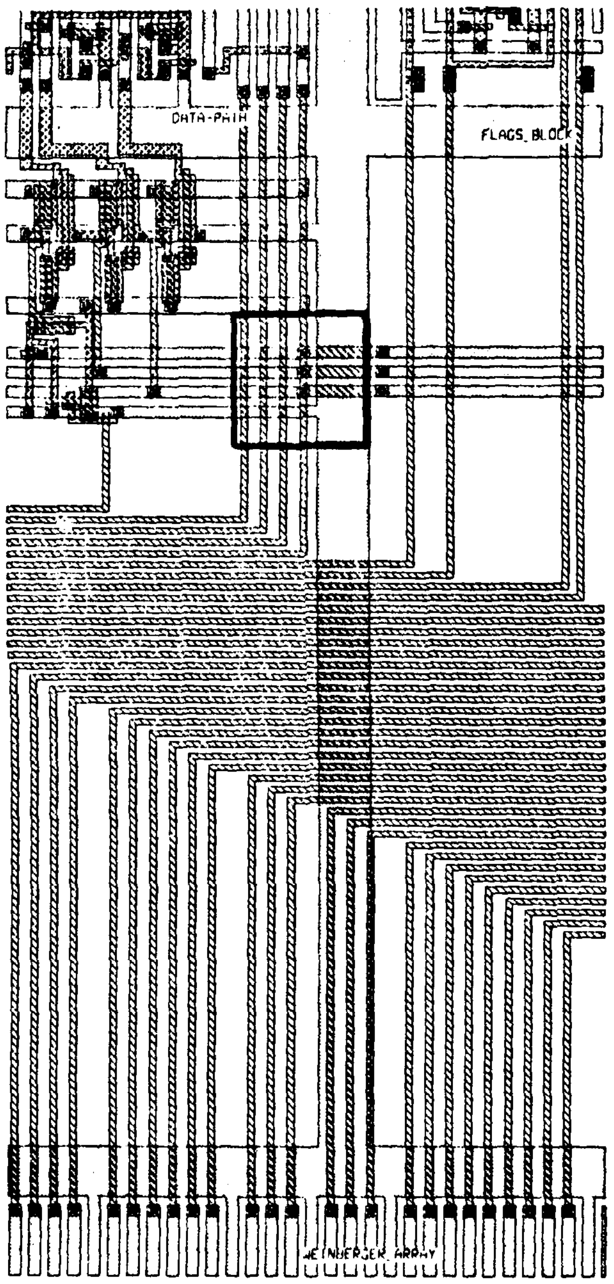


Figure 5.1. MacPitts Wiring Error

line that is always placed on the right edge of the data-path and sequencer. The horizontal clock bus must make metal-to-polysilicon polysilicon-to-metal vias over this ground bus. Since the data wire runs so close to the ground bus it crosses the clock bus at the metal-to-polysilicon via and becomes electrically connected to the clock lines (see Figure 5.2). This error was also found by Kelly (as mentioned in [Ref. 4]) when he used MacPitts to produce a butterfly switching element chip at MIT Lincoln Laboratory. Unfortunately, this error cannot be identified when a design rule check is made on the circuit because no design rules are violated.

It is not difficult to predict when this wiring error is going to occur in the data-path and to correct it when it is found. A programmer should first examine the MacPitts .mac program to identify all statements that cause word size operations to be performed and cause the compiler to produce an organelle in the data-path. If the last word size statement in the .mac program uses the "bit" data-path function of the form:

```
(bit <bit-position> <integer-expression >)
```

the organelle that transfers data from the data-path to the Weinberger array will be placed on the right edge of the data-path and a fatal wiring error will occur. (See [Ref. 2] for a description of the bit function.)

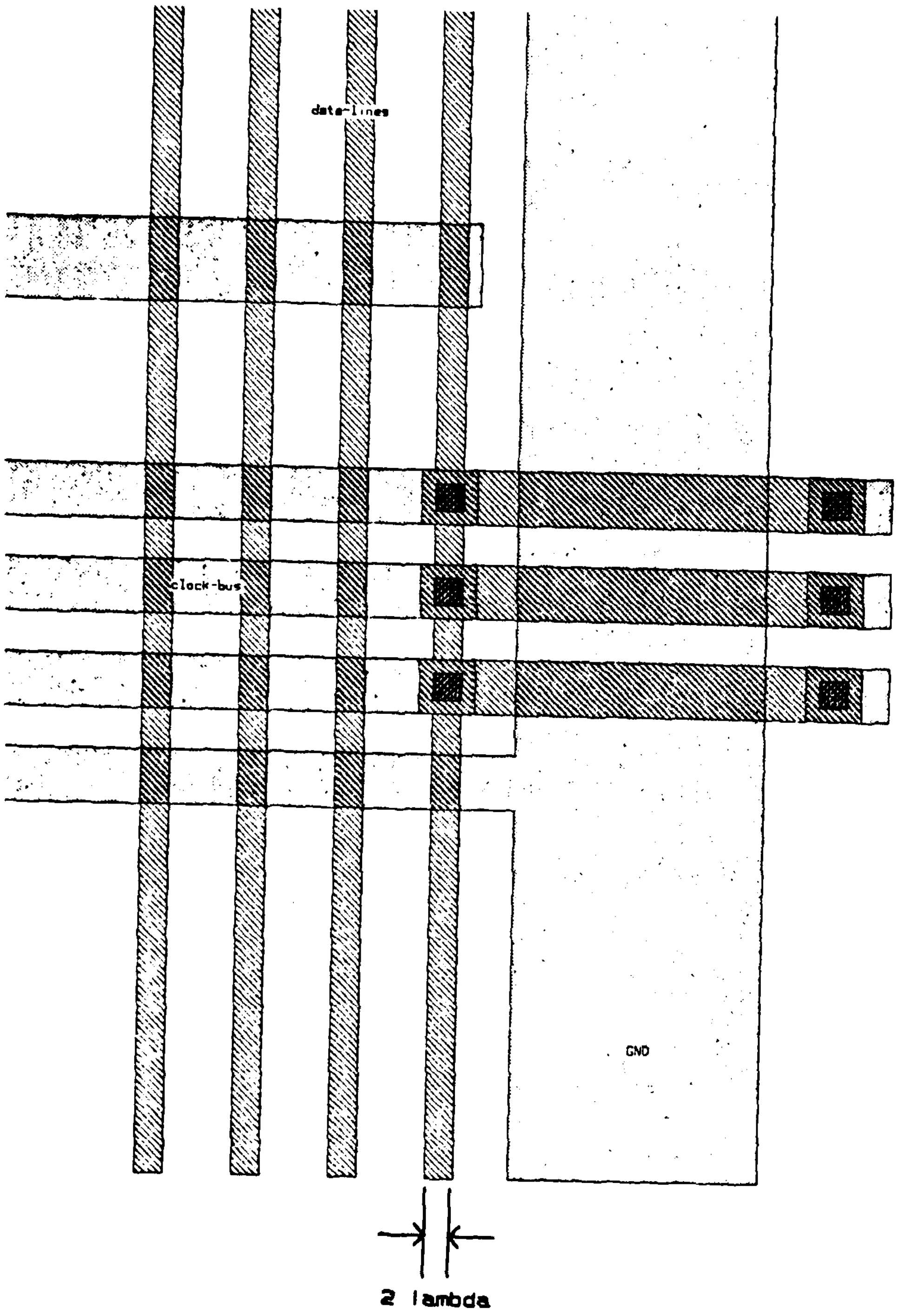


Figure 5.2. Close-up of Wiring Error

It is more difficult to predict when this wiring error is going to occur in the sequencer than in the data-path. Reference 8 contains details of sequencer wiring errors.

2. Correction of Wiring Errors

The wiring errors in the data-path and sequencer can be easily corrected using the Caesar VLSI circuit editor. The Caesar file that contains the last organelle of the data-path or the sequencer must first be identified. This file is then edited using Caesar and the right one or two data lines are rerouted around the clock bus via as shown in Figure 5.3.

If it has been determined that the "bit" function is the last work size statement in the .mac program the steps used in the MacPitts design cycle of a 5 micron design that are listed on page 68 of reference 4 should be modified as follows:

1. Generate a 5 micron .cif file as stated. The following command will create several Caesar files each containing a description of part of the design. (Ignore user extension warning).

   ```
   % cif2ca -1 250 filename.cif
   ```

2. Rename the top level Caesar file.

   ```
   % mv project.ca filename.ca
   ```

3. Use Caesar to identify the Caesar file, symbol xx.ca, that has the wiring in it.

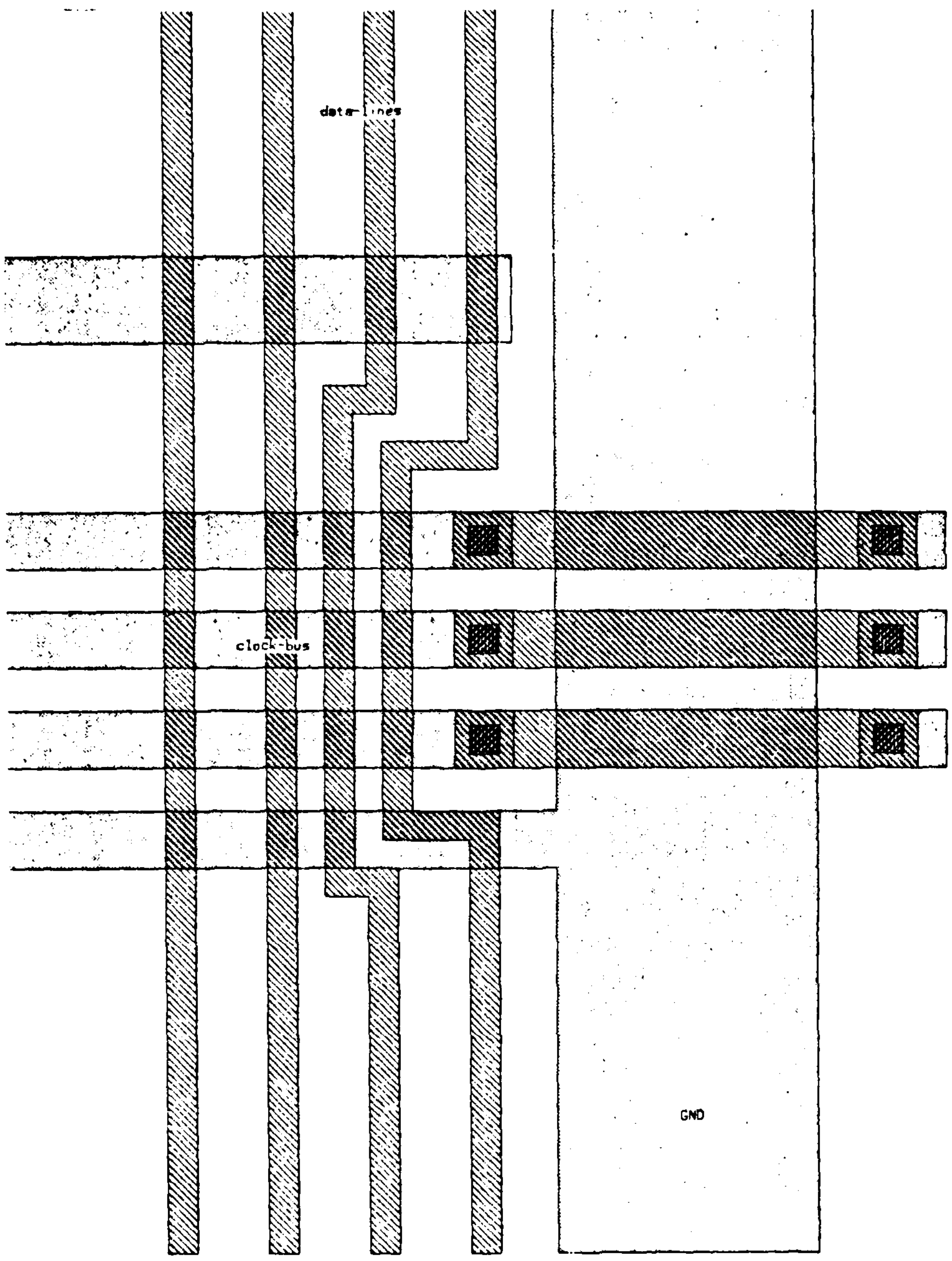


Figure 5.3. Correction of Wiring Error

APPENDIX A

THE MACPITTS INTERPRETER

A. USE OF THE INTERPRETER

The MacPitts interpreter is used to test for syntax and logical errors in the .mac file. The interpreter creates a functional environment of the integrated circuit from the .mac file without actually designing the circuit. This functional environment can then be simulated.

The interpreter can be invoked by using the following command:

```
% macpitts filename int herald
```

Filename is the filename of the .mac file without the .mac extension. Herald is used so that as the MacPitts silicon compiler reaches a milestone as it is processing the .mac file, messages are printed to the terminal. Although the herald statement can be omitted the milestone messages assure the programmer that the silicon compiler is still processing the .mac file on long compile runs.

When the interpreter is ready to start processing a simulation run all registers, ports, processes, flags and signals defined in the .mac file are listed in a table on the terminal screen along with their values (see Figure A.1). The first thirty-six items displayed in the table are labeled from 0-9 and a-z. The MacPitts

9. Redesign the data-path so that data can enter or leave the data-path from either the left or right side to reduce the length of wire runs from the pads.

10. Redesign the flags block and the data-path organelles to save wasted space illustrated in Chapter II.

redesigned to improve circuit speed. A PLA is now used for all chip control functions and the Weinberger array is used only for bit sized boolean functions. The recommended MacPitts improvements listed below, except for #2, #5 and #6, have been incorporated in MetaSyn.

B. RECOMMENDATIONS

The following recommendations should be considered to improve the MacPitts Silicon Compiler:

1. Add super buffers to all input pads.
2. Add super buffers to all data lines leaving the data-path, sequencer and flags block, and to all clock lines driving the registers and flags.
3. Redesign the design from to allow pads on all sides.
4. Use channel routing instead of river routing to reduce the complexity of the Weinberger array.
5. Implement a faster algorithm for design of the Weinberger array.
6. Redesign the registers and flags so that a more conventional two-phase clock can be used in MacPitts designs. This will eliminate the circuit delay of the last three segments of the MacPitts clock that can not be used for logic propagation.
7. Redesign the interpreter to make it more user friendly and able to handle large designs containing many flags, ports, signals, registers and processes as discussed in Appendix A.
8. As mentioned in Chapter III, a data-path organelle should be designed to set and shift data bits of a data word so that data can be transferred from the Weinberger array to the data-path.

attractive alternative when the time required to design a circuit is a more important consideration than the speed or size of the circuit. Until the cause of the alignment errors discussed in Chapter V is found and corrected, all MacPitts designs must be inspected carefully for the possibility of alignment errors. Unexpectedly, it was also found that combinational adder circuits were faster than pipeline adder circuits because of the MacPitts clocking scheme and the timing delay caused by the non-super buffered clock lines driving the registers and flags.

Appendix A gives a complete list of all the MacPitts interpreter commands and an explanation of their use. In addition, all interpreter error statements and their definitions are listed.

In 1983 the developers of the MacPitts silicon compiler (Siskind, Southard, and Crouch [Ref. 3]) left MIT Lincoln Laboratory and formed their own company, MataLogic, Inc., to produce a commercial silicon compiler. MetaLogic's current compiler, called MataSyn, is a redesigned version of the MacPitts compiler. Most of the design limitations of MacPitts have been eliminated in MetaSyn. Two of the more significant improvements in MetaSyn are the redesign of the interpreter and the Weinberger array. The new interpreter, now called the simulator, is very flexible and user friendly and has few of the limitations of the MacPitts interpreter listed in Appendix A. The Weinberger array has been

VI. CONCLUSION

A. SUMMARY

The objectives of this thesis were to determine what the basic circuits in MacPitts designs are and how they are used, to make performance comparisons of several different adder designs with a hand-crafted adder design and to obtain a better understanding of the MacPitts interpreter.

The basic building blocks that the MacPitts compiler uses in circuits were found to be the data-path, the sequencer, the flags block and the Weinberger array. The circuit density and speed of the building blocks were found to be low. This was expected since Siskind was quoted in reference 14 as stating that optimizing chip performance was not a primary MacPitts design goal. The functional description of the circuit in the .mac program was found to have a direct relationship to the circuit structures that the compiler used to design the circuit.

It was found that circuits designed by the MacPitts silicon compiler are very inefficient in terms of the amount of circuitry per chip area and that the speed of a MacPitts circuit is slow compared to hand-crafted designs. The significant advantage that MacPitts-designed circuits have over hand-crafted circuits is the reduction in time required to design the circuit. This makes silicon compilers an compiler was installed under the UNIX 4.1 operating system. It is thought that the version of the Franz LISP compiler installed under UNIX 4.2 may be causing an unexpected roundoff or truncation when the compiler calculates the vertical and horizontal coordinates used to place circuitry on the chip. Alignment errors can be corrected by using Caesar.

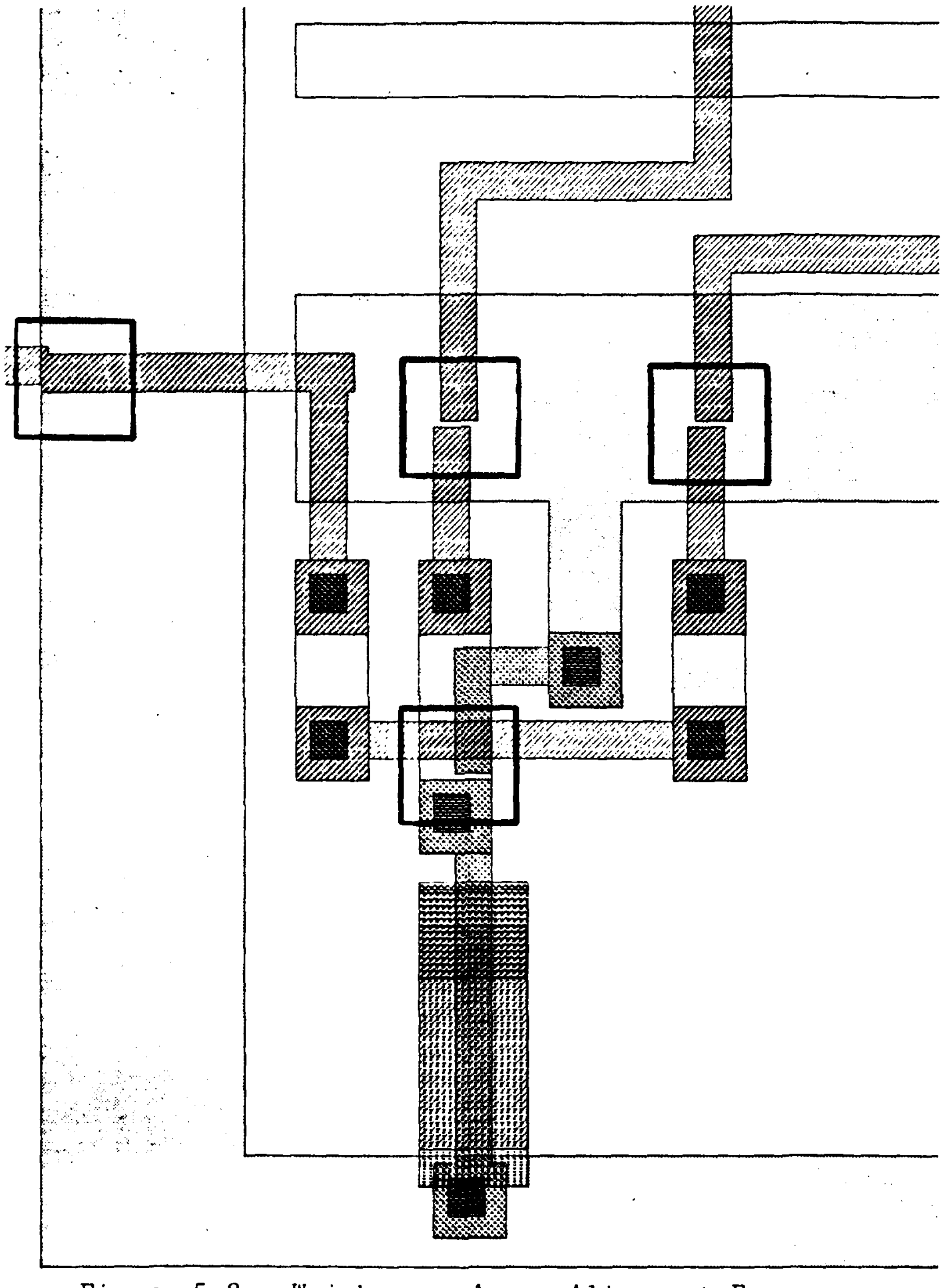


Figure 5.8. Weinberger Array Alignment Errors

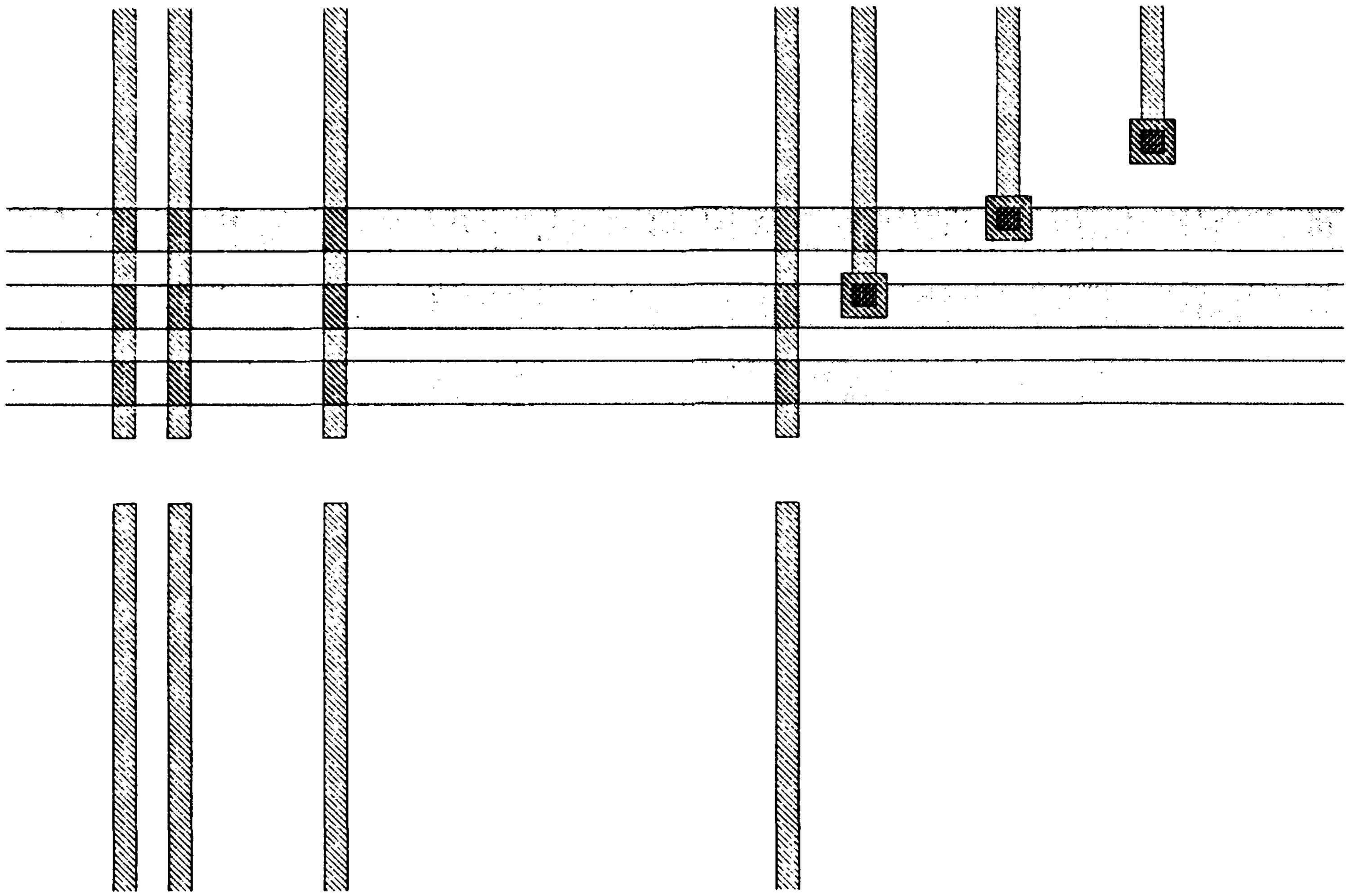


Figure 5.7. Clock and Data Bus Misalignment Errors: 4 Micron 8-Bit 5-Stage Pipeline Adder

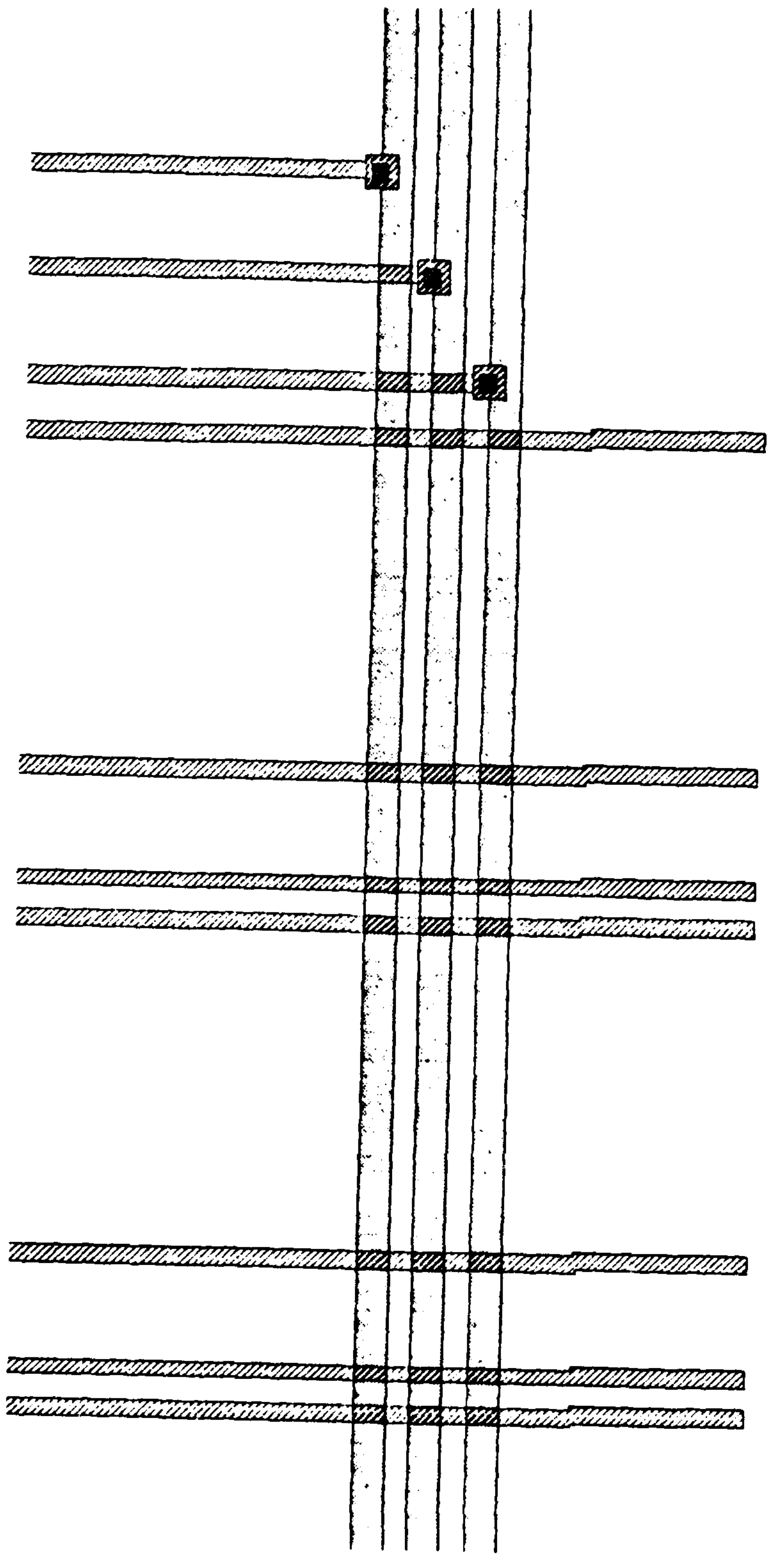


Figure 5.6. Clock and Data Bus Misalignment Errors: 5 Micron 8-Bit Stage Pipeline Adder

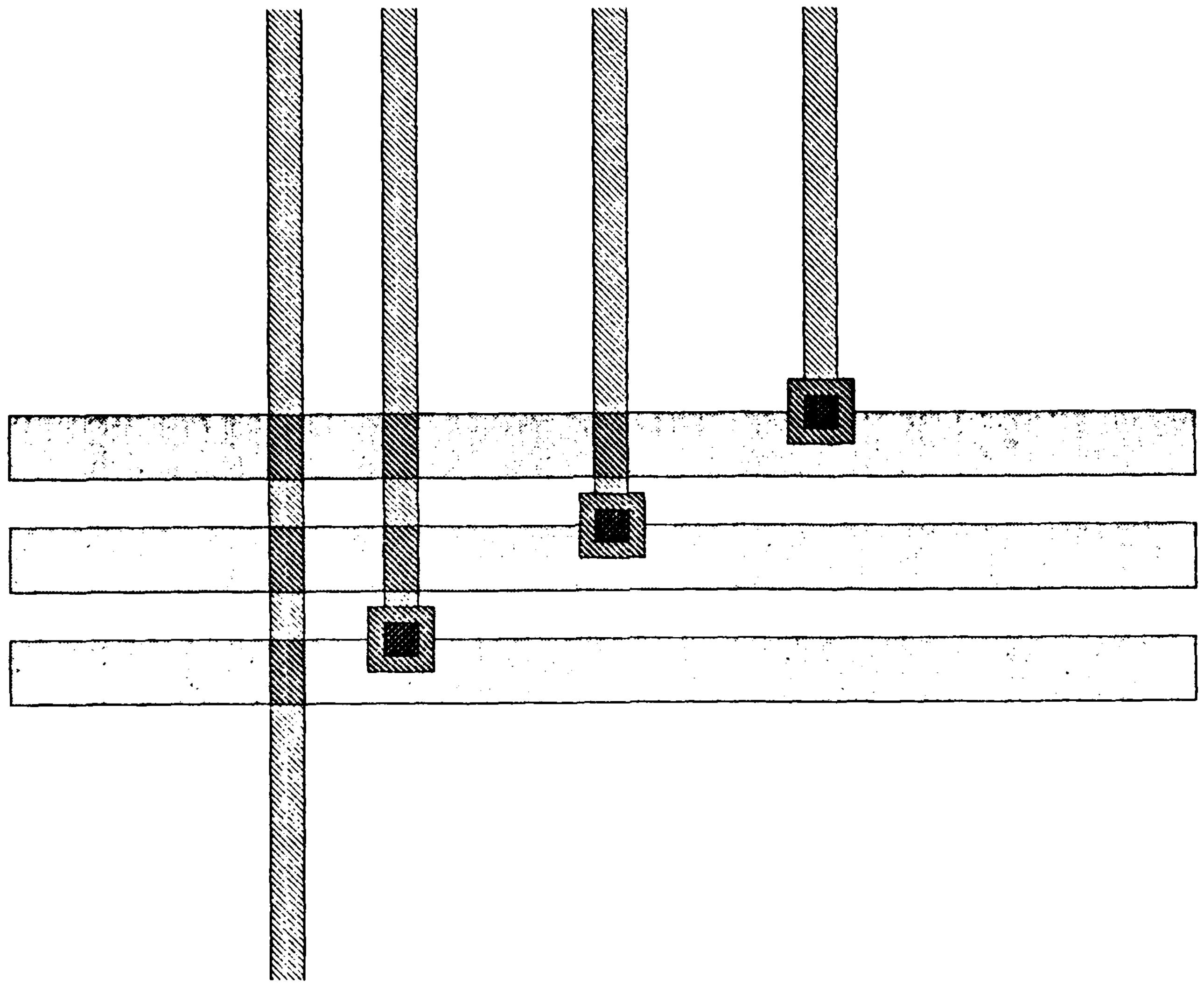


Figure 5.5. Clock Bus Misalignment Error: 4 Micron 4-Bit 5-Stage Pipeline Adder

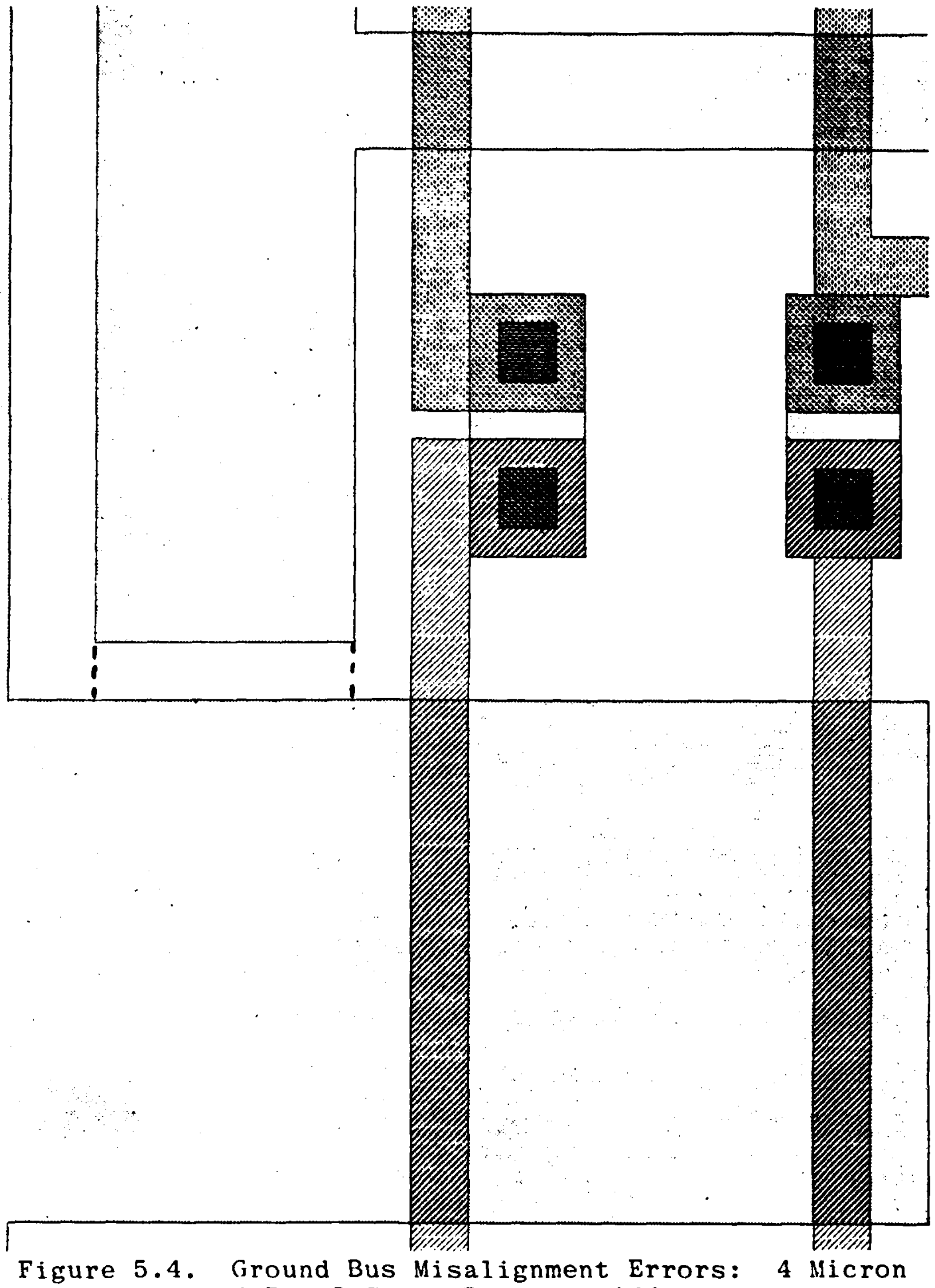


Figure 5.4. Ground Bus Misalignment Errors: 4 Micron 4-Bit 5-Stage Pipeline Adder

the chip clock bus. The flags block of the 5 micron 8-bit 5-stage pipeline adder is placed two lambda too high and one-half lambda too far left on the chip. Figure 5.6 shows the two lambda misalignment of the flags block clock lines and a one-half lambda misalignment of the flags block data lines. The flags block data lines have a two lambda overlap with the chip data lines so even with a two lambda vertical flags block misalignment the data lines are still electrically connected. A flags block misalignment of eight lambda in the vertical direction was found in the 4 micron 8-bit 5-stage pipeline adder. Figure 5.7 shows the clock and data bus alignment errors for this circuit.

The Weinberger array alignment errors are more complex than the flags block errors. In addition to errors where the Weinberger array is placed incorrectly on the chip there are also some internal alignment errors in the array. Figure 5.8 shows three misalignments of the Weinberger array buses and the chip buses. Also shown is one internal misalignment where a diffusion line is not properly connected to a pull-up transistor. Weinberger array alignment errors will be treated in detail in reference 8.

The cause of alignment errors is not yet understood. Alignment errors have only been found in MacPitts designs since the Macpitts compiler was installed under the UNIX 4.2 operating system. No alignment errors were found when the

C. ALIGNMENT ERRORS

Alignment errors have been found in the flags block and the Weinberger array of several different designs. Most of the alignment errors that were found were identified by performing a design rule check on the circuit that contained the errors. The design rule check program is able to find the errors because in most cases metal-to-metal, polysilicon-to-polysilicon or diffusion-to diffusion separation errors occur.

In the flags block the errors have occurred when the compiler places the flags block on the chip so that the internal clock, ground and data buses of the block do not properly align with the chip clock, ground and data buses. The misalignment of the flags block has been found in three designs; the 4 micron 4-bit 5-stage pipeline adder and both the 4 micron and 5 micron 8-bit 5-stage pipeline adders. In each case the circuitry inside the flags block has been designed correctly but the block itself has been placed incorrectly on the chip.

In the case of the 4 micron 4-bit 5-stage pipeline adder the flags block was placed two lambda too high in the circuit. Figure 5.4 shows that the flags block ground bus does not properly connect with the chip ground bus. In Figure 5.5 the metal-polysilicon contacts of the flags block clock lines do not properly align with the metal lines of

```
% caesar filename
```

The Caesar file for the complete data-path/sequencer may have to be edited in Caesar to identify the file that contains the last organelle fo the data-path/ sequencer where the wiring error is located. Caesar can be used to reroute the data lines around the clock bus via.

4. Edit the top level Caesar file again and create a new .cif file.

```
: sa

: cif 248

: q
```

5. Next, perform a design rule check of the new .cif file. (Note that the cif command line ends in -qnq not -gng in the following command).

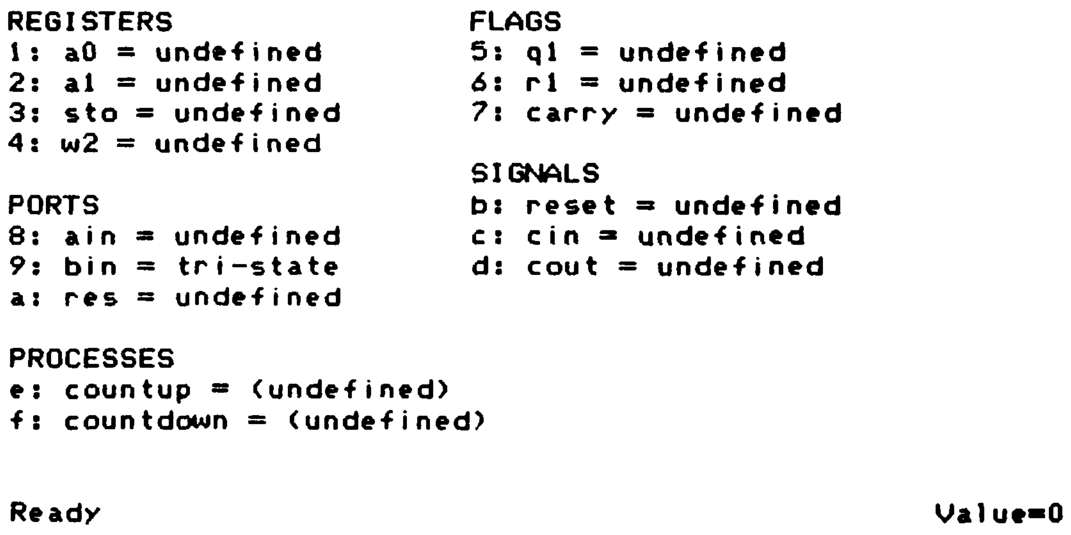
```
% cif filename.cif -qnq

% cll filename.co

% drc filename.sco
```

6. To perform an event simulation on the modified 5 micron design the procedure listed on page 71 of reference 4 for the 4 micron design should be followed to affix labels to the bonding pads, obtain a node extract, and start the simulation run. Insure that the 248 scale is used when creating a new .cif file of a 5 micron design in Caesar (see page 96 of reference 4).

For a micron design that contains wiring errors the MacPitts design cycle listed on page 70 of reference 4 should be followed. The wiring errors can be corrected, using the above procedure, at the same time that the labels are affixed to the bonding pads.



```
REGISTERS                 FLAGS
1: a0 = undefined         5: q1 = undefined
2: a1 = undefined         6: r1 = undefined
3: sto = undefined        7: carry = undefined
4: w2 = undefined
                          SIGNALS
PORTS                     b: reset = undefined
8: ain = undefined        c: cin = undefined
9: bin = tri-state        d: cout = undefined
a: res = undefined

PROCESSES
e: countup = (undefined)
f: countdown = (undefined)

Ready                                            Value=0
```

Figure A.1. The Interpreter Screen Display

interpreter does not have the ability to label more than thirty-six items so items thirty-seven and higher are not labeled. At the bottom of the screen a command line is displayed. The command line shows the status of the interpreter at any time. Possible command line displays are Ready, indicating that the interpreter is ready to accept a command, and Clocking, indicating that the interpreter is performing a functional simulation of the chip through one or more clock cycles. On the bottom right of the screen the contents of a special interpreter register called "value" are shown. The value register is used to set ports and registers to particular values and also indicates the number of clock cycles a simulation run will execute.

There is one serious limitation with the interpreter that causes it to be unusable for many large chip designs. If the total number of registers, ports and processes defined in the .mac file is greater than twenty there will be too many items for the interpreter to display on the right side of the terminal screen at once (see Figure A.1). Also, if the total number of flags and signals is greater than twenty-two there will be too many items for the interpreter to display on the left side of the terminal screen at once. Unfortunately, the interpreter continues to try to display those items that will not fit on the screen. Since the interpreter is never able to display all items control of the terminal is never turned over to the programmer for a simulation run. The only way to stop the interpreter if this happens is to abort the run by typing a control Z.

The interpreter uses information from three different locations to determine the values of all registers, ports, signals, flags and processes during a simulation run. The first location is the "console" where the programmer, using the terminal keyboard, can specify the values of the above items. The second location is the functional environment of the circuit, called the "chip". This is where the interpreter uses input information from the programmer to determine the values of the above items. The last location is called the "environment" and is a programmer specified functional environment that the programmer may have the interpreter use during simulation (see the "e" command below).

B. INTERPRETER COMMANDS

All interpreter commands are screen oriented which means the command is executed as soon as the key is pressed and a carriage return is not necessary. Table VI gives a list of the interpreter commands. These commands can be displayed on the screen by typing "?".

Most of the interpreter commands are self-explanatory but several require additional explanation. Several commands

TABLE VI

MACPITTS INTERPRETER COMMAND SUMMARY

```
         ? - This menu
        ^L - Repaint screen
         p - Put interpreter state to <file-name>.int
         g - Get interpreter state from <file-name>.int
         e - Enable/Disable environment from <file-name>.env
         c - Clock system <value> cycle(s)
         l - Escape to Lisp system
         q - Quit
         j - Move cursor down
         k - Move cursor up
   G <tag> - Move cursor to <tag>
         t - Set flag, input signal, or i/o signal to t
         f - Set flag, input signal, or i/o signal to f
         s - Set register, input port, or i/o port to <value>
         u - Set register, flag, input port, i/o port,
                input signal, or i/o signal to undefined
         T - Set i/o port or i/o signal to tri-state
         x - Clear <value> register to 0
         - - Negate <value> register
   <digit> - Enter <digit> into <value> register
```

affect only one item on the screen. Before these commands can be used the item to be affected must be highlighted by the inverse cursor. The "j", "k" and "G" commands are used to move the cursor around the screen. If an adm3a terminal is used instead of a vt100 terminal the inverse cursor is not displayed and only the "G" command can be used to place the "invisible" cursor over the item to be affected.

When the registers, ports, processes, flags and signals are initially displayed on the screen by the interpreter their values are undefined or tri-state if a tri-state port or signal is defined in the .mac file. (See Reference 2 for an explanation of the different register, port and signal types.) Before a simulation run is made all input and i/o ports and signals must be set to some initial value. The "t" and "f" commands are used to set input or i/o signals to true or false, respectively. The "s" command is used to set an input or i/o port to the value stored in the value register. Another command, the "T" command, can also affect the values of input or i/o ports and signals but has proven to be not very useful. If the "T" command is used on an input port or signal, or an i/o port or signal that is used for input only in the .mac program, the port or signal value will be set to a high impedance state (tri-state). The port or signal value will stay at high impedance until explicitly set to some value by the programmer using the "s", "t", or "f" commands. If an i/o port or signal is used for output only or both input and output in the .mac program the "T" command will cause the port or signal value to change to undefined.

The "c" command is used to simulate the functional environment of the chip. The number of clock cycles simulated in one simulation run is indicated by the value register. If 0 or 1 is stored in the value register only one clock cycle will be simulated.

After a simulation run it may be desirable to store the values of all items displayed on the screen. This can be done by using the "p" command. The state of the functional environment is saved in a file called filename.int where filename is the same as the filename.mac file. If more than one state is to be saved the programmer must login on another terminal and rename the .int file after each state is saved because each new state will be saved in the same .int file.

The programmer also has the option of specifying the functional environment that the interpreter will use to simulate a particular .mac file [Ref. 2]. The "e" command is used to enable/disable a functional environment stored in the filename.env file. There is no published information or documentation on the format of the functional environment in the .env file so this option has never been used at the Naval Postgraduate School.

C. INTERPRETER ERROR STATEMENTS

As stated before, the purpose of performing an interpreter simulation is to check for syntax and logical errors in the .mac file before a full chip design is made by the MacPitts compiler. Logical errors can be found by performing a simulation run and then comparing the results obtained to those expected. Reference 4, pages 47-49, shows a good example on how to perform a simulation of a .mac file using the MacPitts interpreter.

Syntax errors in the .mac file are indicated in one of two ways by the compiler. First, if the error is severe enough the compiler stops the creation of the functional environment and displays an error message on the terminal screen that will give an indication of the syntax error. The compiler then returns the UNIX operating system back to the programmer. An example of a severe syntax error is an unequal number of open and closed parentheses in the .mac file. Less severe syntax errors usually do not show up until initial values are loaded into the input or i/o ports and signals or until a simulation run is performed. A short error message is then displayed on the command line of the terminal screen.

There are over thirty different error messages that the compiler can display when a syntax error is found. The error messages and their meaning are as follows:

1. Interpreter error 1: the interpreter tried to change the state (value) of a register but found the current state to be empty (null), possessing no value. This error indicates an improper register definition or usage in the .mac file.
2. Interpreter error 2: same as 1 above but for a flag.
3. Interpreter error 3: same as 1 above but for a port.
4. Interpreter error 4: same as 1 above but for a signal.
5. Interpreter error 5: Unrecognizable function. Examples of some expected functions are setq, not, bit, call and if. See reference 2 for a listing of all MacPitts functions.
6. Interpreter error 6: the antecedent of an if statement is not t, f or undefined as required.
7. Interpreter error 7: the interpreter tried to determine the state (value) of a register but found the current state to be empty (null), possessing no value. This error indicates an improper register definition or usage in the .mac file.
8. Interpreter error 8: same as 7 above but for a flag.
9. Interpreter error 9: same as 7 above but for a process.
10. Interpreter error 10: same as error 7 above but for a port.
11. Interpreter error 11: same as error 7 but for a signal.

12. Unrecognizable atomic form: an unknown alpha-numeric string is in the .mac file. Check for a missing definition or misspelled word.

13. Process state out of bounds: the state (value) of a process is less than zero. Check the .mac file for a statement improperly setting a process to a value less than zero.

14. This process has too many returns: a return from a subroutine was encountered for which there was no previous call statement. Check the .mac file for the correct number of returns or for a missing call statement.

15. This process has too many calls: a call to a subroutine was made but no return statement was found. Check .mac file for correct number of calls or for a missing return statement.

16. Invalid bit selector: the bit selector in the data-path function "bit" is not between 0 and the bit size of the data-path as required.

17. Too many arguments: all MacPitts functions require only one or two arguments. Check the .mac file and Reference 2.

18. Too few arguments: see 17 above.

19. A reset signal is needed: a reset signal has not been defined when the "process" form is used in the .mac file.

20. Double signal (port) setq, chip vs. environment: the interpreter attempts to set a signal (port) to a value different than that assigned to that signal (port) by the functional environment from the .env file.

21. Double signal (port) setq, chip vs. console: the interpreter attempts to set a signal (port) to a value different than that assigned to that signal (port) by the programmer using the "s", "t" or "f" commands.

22. Double signal (port) setq, environment vs. chip: the reverse of 20 above.

23. Double register setq: two different setq statements in the .mac file attempt to assign a value to the same register at the same time.

24. Double process setq: same as 23 above but for a process.

25. Double port setq: same as 23 above but for a port.

26. Only one character per character-constant: this error indicates that an attempt was made to set the value of a constant to a character string longer than one character. The value of a constant can be an integer or a single character. If the value of a constant is set to a single character the ASCII equivalent of that character becomes the value of the constant.

In addition to the above syntax error statements there are two syntax warning statements. These statements indicate that there may be a syntax error and caution should be exercised during the simulation. These warning statements are:

1. This process has undefined state: the interpreter has encountered a process in the functional environment whose state (value) is undefined.
2. Antecedent of if is undefined: the interpreter has encountered a register, port, signal, process, or flag in the functional environment being used as the antecedent of an if statement and whose value is undefined.

The above two warning statements are common for pipeline design architectures. Initially the value of the ports, registers, processes, signals, and flags of each stage of the pipeline are undefined and will stay undefined until data is clocked into and out of each stage.

The MacPitts interpreter also displays error statements in the command line of the terminal screen if an interpreter command has been executed improperly by the programmer. The interpreter command error statements are:

1. File not found: .int or .env file cannot be found.
2. Cannot set this thing to value: only registers and ports can be set to value.
3. Cannot set this thing to t, f: only signals or flags can be set to t or f.

4. Cannot set this thing to undefined: processes cannot be set to undefined.
5. Cannot set this thing to tri-state: only input or i/o ports and signals can be set to tri-state.
6. Invalid command type ? for help: check interpreter command list for correct command.
7. Cannot input from this port (signal): check for input or i/o port (signal).
8. Cannot output to this port (signal): check for output or i/o port (signal).

LIST OF REFERENCES

1. The Microelectronics Center of North Carolina Technical Report 83-06, Silicon Compilers: A Critical Survey, by R. R. Gross, 31 May 1983.

2. Lincoln Laboratory, Massachusetts Institute of Technology Project Report RVLSI-3, An Introduction to MacPitts, J. R. Southard, 10 February 1983.

3. Siskind, J. M., Southard, J. R. and Crouch, K. W., "Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions", Proceedings, Conference on Advanced Research in VLSI, pp. 28-40, 1981.

4. Carlson, D. J., Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers, Master's Thesis, Naval Postgraduate School, Monterey, CA, June, 1984.

5. Conradi, J. R. and Hauenstein, B. R., VLSI Design of a Very Fast Pipelined Carry Look Ahead Adder, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1983.

6. "Crystal Users Manual", In University of California at Berkeley Report No. UCB/CSD 85/225, Berkeley VLSI Tools: More Works by the Original Artists, by Scott, W. S., Hamachi, G., Ousterhout, J. and Mayo, R. N., February 1985.

7. Department of Electrical and Computer Science, Massachusetts Institute of Technology VLSI Memo No. 83-132, Optimization of the MacPitts Silicon Compiler for Telecommunications Circuitry, J. R. Fox, January 1983.

8. Larrabee, R. C., unpublished, Master's Thesis, Naval Postgraduate School, Monterey, CA, expected date of publication, September 1985.

9. Chen, T. C., "Overlap and Pipeline Processing", An Introduction to Computer Architecture, edited by H. S. Stone, Science Research Associates, Inc., 1980.

10. Hwang, K., Computer Arithmetic Principles, Architecture, and Design, J. Wiley & Sons, 1979.

11. Ousterhout, J., "Using Crystal for Timing Analysis", In University of California at Berkeley Report No. UCB/CSD 85/225, Berkeley VLSI Tools: More Works by the Original Artists, by W. S. Scott, G., Hamachi, J. Ousterhout and R. N. Mayo, February 1985.

12. Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley, 1980.

13. Newkirk, J. A. and Mathews, R., The VLSI Designer's Library, Addison-Wesely, 1983.

14. Werner, J., "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?", VLSI Design, v. 3, pp. 46-52, September/October 1982.

BIBLIOGRAPHY

Southard, J. R., "MacPitts: An Approach to Silicon Compilation", Computer, v. 16, pp. 74-82, December 1983.

Weinberger, A., "Large Scale Integration of MOS Complex Logic: A Layout Method", IEEE Journal of Solid State Circuits, v. sc-2, pp. 182-190, December 1967.

Winston, P. H. and Berthold, K. P. H., LISP, 2d ed., Addison-Wesley, 1984.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, Virginia 22314-6145

2. Superintendent — 2
   Attn: Library, Code 0142
   Naval Postgraduate School
   Monterey, California 93943-5100

3. Dr. Donald Kirk — 5
   Code 62Ki
   Naval Postgraduate School
   Monterey, California 93943-5100

4. Dr. H. H. Loomis — 2
   Code 62Lm
   Naval Postgraduate School
   Monterey, California 93943-5100

5. Chairman, ECE Department — 2
   Code 62
   Naval Postgraduate School
   Monterey, California 93943-5100

6. CPT Alexander O. Froede, III — 1
   2914 Bayeux Avenue
   Melbourne, Florida 32935

7. Dr. Antun Domic B-347 — 1
   Massachusetts Institute of Technology
   Lincoln Laboratory
   P. O. Box 73
   Lexington, MA 02173-0073

8. Mr. Jay Southard — 1
   MetaLogic Inc.
   725 Concord Avenue
   Cambridge, MA 02138